P. 1877/87

8 1987

informatyka

Grafika w Turbo Pascalu

Prolog w bazach danych

Cykl o języku C

WYDAWNICTWO NOT SIGMA

Cena 150 zł

informatyka

8 1987

Nr 8

Miesięcznik Rok XXII

Sierpień 1987

Organ Komitetu
Naukowo-Technicznego NOT
ds. Informatyki

KOLEGIUM REDAKCYJNE:

Mgr Jarosław DEMINET, dr inż. Wacław ISZKOWSKI, mgr Teresa JABŁOŃSKA (sekretarz redakcji), Władysław KLEPACZ (redaktor naczelny), dr inż. Marek MACHURA, Maria PAWLAK (sekretarz redakcji), dr inż. Wiktor RZECZKOWSKI, mgr inż. Jan RYŻKO, mgr Hanna WŁODARSKA, dr inż. Janusz ZALEWSKI (zastępca redaktora naczelnego)

PRZEWODNICZĄCY
RADY PROGRAMOWEJ:

Prof. dr hab. Juliusz Lech Kulikowski

Materiałów nie zamówionych redakcja nie zwraca

Redakcja: 00-517 Warszawa, ul. Mickiewicza 18 m. 17, tel. 39-14-34

Zakł. Graf. „Tamka". Zam. 0488-1300/87. Obj. 4,0 ark. druk. Nakład 8200 egz. K-79.

ISSN 0542-9951, INDEKS 36124

Cena egzemplarza 150 zł
Prenumerata roczna 1800 zł

WYDAWNICTWO
NACZELNA ORGANIZACJA TECHNICZNA
CZASOPISM I KSIĄŻEK TECHNICZNYCH
SIGMA
00-950 Warszawa
skrytka pocztowa 1004
ul. Biała 4

W NUMERZE:

	Strona
Zastosowanie Prologu w bazach danych (1) *Włodzimierz Grudziński*	1
Podstawy grafiki w języku Turbo Pascal (1). Przegląd środków graficznych *Jan Bielecki*	5
Poza programowanie wielkoskalowe — kolejne wyzwania dla inżynierii oprogramowania (2) *Mary M. Shaw*	8
Głosowe wprowadzanie informacji do komputera (2) *Ryszard Tadeusiewicz*	12
Struktura systemu operacyjnego PC-DOS (2) *Andrzej Raznowiecki, Anna Syfert, Janusz Zalewski* (oprac.)	16
Język C — definiowanie funkcji ze zmienną liczbą argumentów *Jan Bielecki*	19
Norma ANSI języka C oprac. *M. Kuc*	20
RECENZJE	24
Drugie wejście Codda	
ZE ŚWIATA	25
Programowanie w Adzie rozproszonych systemów docelowych Pierwszy mikrokomputer z mikroprocesorem 80386	
POGLĄDY	23
Logo Sinclaira i jego modyfikacje — przyczynek do dyskusji o jakości pozasprzętowej bazy szkolnej edukacji informatycznej	
TERMINOLOGIA	31

W NAJBLIŻSZYCH NUMERACH:

- Gerhard Goos o produkcji oprogramowania
- Piotr Zielczyński o systemie MULTICOMP do rozwiązywania zadań metodą przeszukiwania drzew
- Stanisława Ossowska o zasadzie działania i sposobie użytkowania Wordstara 3.30
- Danuta Grabowicz o wprowadzeniu do programowania w języku assemblera IBM PC
- Jan Bielecki o wykreślaniu podstawowych obiektów graficznych w języku C
- Mariusz Kuc o konwersji kodowania w języku Ada
- Zbigniew Szkaradnik o procesorze Novix NC 400

WŁODZIMIERZ GRUDZIŃSKI
Instytut Informatyki
Uniwersytet Warszawski

P. 1877/87

POLSKIE TOWARZYSTWO INFORMATYCZNE

Pierwsza wersja referatu została przedstawiona podczas Szkoły Jesiennej PTI „Współczesne kierunki rozwoju informatyki" — Mrągowo, 3—7 listopada 1986 r.

Zastosowanie Prologu w bazach danych (I)

W ostatnich latach nastąpił szybki rozwój komputeryzacji różnych dziedzin działalności gospodarczej, administracyjnej, dydaktycznej, wojskowej i naukowej. Nowe potrzeby użytkowników oraz rozmaitość oferowanego sprzętu spowodowały wzrost wymagań stawianych oprogramowaniu. Dotyczy to także systemów baz danych coraz częściej stosowanych nie tylko do tradycyjnie podejmowanego przetwarzania danych, lecz także do wspomagania projektowania, automatyzacji prac biurowych, w systemach ekspertowych (ang. expert systems) itp. Po pierwszym okresie rozwoju systemów baz danych, w trakcie którego zasadniczo uporano się z problemami sprawnego zarządzania dużymi zbiorami danych, efektywnej realizacji żądań użytkowników, niezawodności oraz ochrony dużych baz danych, obecnie poszukuje się przede wszystkim lepszych sposobów modelowania świata rzeczywistego oraz zwiększenia „inteligencji" systemów zarządzania bazami danych. Do nowych wymagań stawianych przed tymi systemami należą między innymi:

— zapewnienie obsługi obiektów o złożonej strukturze (szczególnie utrudnione w systemach relacyjnych),

— obsługa niesformatowanej informacji, np. tekstów, obrazów, map, sygnałów dźwiękowych,

— przechowywanie pewnej wiedzy o modelowanym świecie wyrażonej nie tylko za pomocą faktów (danych), lecz także w postaci ogólnych reguł,

— umiejętność przynajmniej prostego wnioskowania opartego na zapamiętanych regułach i faktach.

Wiele problemów związanych z dwoma ostatnimi zagadnieniami wchodzi w zakres badań nad sztuczną inteligencją. Jednym z najbardziej obiecujących podejść do ich rozwiązania jest wykorzystanie wyników oraz narzędzi stosowanych w nowej, powstałej w początkach lat siedemdziesiątych dziedzinie informatyki, zwanej programowaniem w logice [1]. Zainteresowanie programowaniem w logice wzrosło gwałtownie w ciągu ostatnich kilku lat, zwłaszcza po przyjęciu go za podstawę oprogramowania systemowego w japońskim projekcie opracowania tzw. piątej generacji komputerów.

Celem tego artykułu jest przedstawienie dotychczas osiągniętych wyników zastosowania programowania w logice, a w szczególności języka Prolog, w systemach baz danych. Mówiąc o bazach danych autor ma na myśli relacyjny model danych i relacyjne systemy baz danych, choć większość rozważań odnosi się również do innych modeli i systemów. Należy podkreślić, że prowadzone do tej pory prace mają charakter badawczy i na praktyczne (komercyjne) zastosowania trzeba będzie jeszcze trochę poczekać. Artykuł jest opisowy i przeglądowy, dlatego prezentacja tematu nie jest formalna.

[1]) Takie sformułowanie pozostawiono na wyraźne życzenie Autora — powinno być „programowanie w języku logiki" lub „programowanie logiczne" — p. dyskusja w rubryce terminologicznej na str. 31 (przyp. red.).

Mgr WŁODZIMIERZ GRUDZIŃSKI ukończył w 1974 r. Wydział Matematyki i Mechaniki Uniwersytetu Warszawskiego. Obecnie pracuje w Ośrodku Obliczeniowym Instytutu Informatyki Uniwersytetu Warszawskiego. Zajmuje się problematyką związaną z systemami baz danych i programowaniem w logice.

PROGRAMOWANIE W LOGICE

Przez wiele lat logika pierwszego rzędu była stosowana w badaniach nad sztuczną inteligencją jako formalizm do reprezentacji wiedzy i rozwiązywania problemów. Również w bazach danych, zwłaszcza w teorii relacyjnych baz danych, stosowano logikę w pracach dotyczących języków zapytań, wyznaczania odpowiedzi na zapytania, projektowania schematów baz danych, tworzenia i utrzymywania więzów integralnościowych itp. [21], [5]. R. Kowalski [14] wykazał, że logikę można traktować jak język programowania. Programy mogą być wyrażone jako zdania logiki pierwszego rzędu, uzupełnione odpowiednimi informacjami opisującymi sterowanie, tj. sposób wykonania programu. Obliczenie polega wówczas na kierowanym poszukiwaniu dowodu.

Każde zdanie logiki pierwszego rzędu może być przekształcone na zbiór klauzul postaci:

$$A_1 \vee \ldots \vee A_n \leftarrow B_1, \ldots, B_m \qquad (1)$$

gdzie $\vee$ oraz, (przecinek) oznaczają odpowiednio alternatywę i koniunkcję, a $\leftarrow$ implikację. A_i i B_j ($i = 1, \ldots, n$, $j = 1, \ldots, m$) są literałami postaci $p(t_1, \ldots, t_k)$, gdzie p jest nazwą predykatu o krotności k, a każdy term t_i może być stałą, zmienną lub funktorem, po którym następuje lista termów (argumentów). Przykładami literałów są:

para (1,2)
xyz (X,Y,Z)
lista (Pocz,Kon)
równanie (X, plus (1, minus (5,a)))
$\sim$ab (a,b), przy czym $\sim$ oznacza zaprzeczenie.

Język programowania w logice traktuje się czasami jako podzbiór logiki pierwszego rzędu, zwany klauzulami Horna. Klauzule Horna mają postać implikacji:

$$A \leftarrow B_1, \ldots, Bm \qquad (2)$$

gdzie B_i są nazywane przesłankami, a A wnioskiem. Jeśli brak jest przesłanek, to klauzula (2) ma postać stwierdzenia A (tj. zdania $A \leftarrow$), a jeśli nie ma wniosku — to ma postać zaprzeczenia $\sim(B_1, \ldots, B_n)$ (zdanie $\leftarrow B_1, \ldots, B_n$).

Można powiedzieć, że predykat odpowiada pewnej relacji zachodzącej między obiektami pewnego świata. Przyporządkowanie obiektom nazw predykatów, funktorów, stałych i zmiennych wyznacza interpretację zdania (klauzuli) logiki. O znaczeniu klauzuli można mówić dopiero po ustaleniu odpowiedniej interpretacji.

Zakłada się również, że wszystkie zmienne występujące w klauzulach są w zasięgu kwantyfikatora ogólnego. Na przykład, jeśli predykatom i zmiennej X nada się naturalną interpretację, to klauzula:

ata (X) $\vee$ biega (X) $\leftarrow$ ptak (X) (3)

oznacza, że wszystkie ptaki potrafią latać lub biegać. We wszystkich przykładach będzie stosowana, przyjęta na ogół w implementacjach Prologu, konwencja rozpoczynania nazw zmiennych dużą literą, a stałych, predykatów i funktorów — małą.

Jeśli dana jest interpretacja, to można stwierdzić, czy spełnia ona pewien zbiór klauzul. Dokonuje się tego wy-

znaczając wartości klauzul, którymi mogą być „prawda" lub „fałsz". Mówimy również, że klauzula **w** wynika ze zbioru klauzul **W**, jeśli każda interpretacja spełniająca **W** spełnia również **w**. Na przykład ze zbioru zdań:

{ptak (orzeł), ∼ biega (orzeł)}

i klauzuli (3) wynika:

lata (orzeł)

Do formalnego wnioskowania, umożliwiającego wyprowadzanie zdań ze zbioru innych zdań jedynie na podstawie ich syntaktycznej postaci, służą reguły wnioskowania. Do najbardziej znanych należy reguła modus ponens i prawa De Morgana. Języki programowania w logice korzystają zazwyczaj z jednej reguły wnioskowania, zwanej rezolucją [19].

Reguła rezolucji mówi, że z dwóch zdań postaci:

$\sim X \vee Y$ (czyli $Y \leftarrow X$) i $X \vee Z$

wynika zdanie:

$Y \vee Z$

Warto przyjrzeć się najprostszym i szczególnie interesującym przypadkom zastosowania tej reguły.

1. Z dwóch klauzul:
zaprzeczenia S1: **∼A**
oraz implikacji S2: **A ← B**
zastosowanie rezolucji wyprowadzi nową klauzulę:
s: **∼B**

2. Z klauzul:
zaprzeczenia S1: **∼A**
stwierdzenia S2: **A**
otrzymamy klauzulę pustą, co oznacza, że doprowadzono do sprzeczności.
Oczywiście w obu przypadkach predykaty **A** w S1 i S2 muszą być identyczne.

Przypadek 2 jest znany w matematyce jako sprowadzenie do sprzeczności i tak właśnie stosuje się rezolucję w językach programowania w logice. Zatem, chcąc wykazać, że klauzula **w** wynika z **W**, należy wziąć zaprzeczenie **w** i znaleźć w zbiorze **W** klauzulę, której wniosek jest identyczny z **w** lub da się do takiego sprowadzić za pomocą odpowiedniego przekształcenia. Przekształcenie to polega na znalezieniu tzw. unifikatora głównego, tj. właściwego podstawienia na zmienne. Istnieje algorytm, zwany algorytmem unifikacji lub uzgadniania, który dla każdego skończonego zbioru wyrażeń znajduje jego unifikator główny lub sygnalizuje, że jest to niemożliwe. Z jego pomocą można również uzyskać ciąg kolejno znajdowanych unifikatorów głównych. Następnie należy zastosować regułę rezolucji do tej wybranej klauzuli i do **∼w**. Otrzymana klauzula jest dodawana do zbioru **W** i powyższy proces powtarza się dopóty, dopóki nie otrzyma się klauzuli pustej lub stwierdzi niemożliwość znalezienia odpowiedniej klauzuli w **W**. Wynika stąd, że ta metoda pozwala nie tylko wykazać sprzeczność pewnego zbioru klauzul, czyli odpowiedzieć na pytanie o prawdziwość **w**, lecz także znaleźć obiekty powodujące tę sprzeczność. Oczywiście należy przyjąć pewien sposób wyboru klauzul ze zbioru **W**.

Następujący przykład ilustruje sposób użycia tej metody do formalnego wnioskowania. Niech będą dane klauzule:

ojciec (jan,piotr)
ojciec (piotr,anna)
ojciec (jan,ewa)

opisujące pewną genealogiczną bazę danych.

Można wówczas zadawać następujące rodzaje pytań:

1. Czy Jan jest ojcem Piotra?
Aby uzyskać odpowiedź na to pytanie należy sprawdzić, czy z powyższego zbioru klauzul wynika:
∼ojciec (jan,piotr)
Stosując metodę rezolucji otrzymuje się natychmiast klauzulę pustą. To znaczy, że powyższa klauzula jest sprzeczna ze zbiorem danych klauzul, czyli odpowiedzią na pytanie jest „tak".

2. Kto jest ojcem Piotra?
Tym razem bada się wynikanie klauzuli:
∼ojciec (X,piotr)
gdzie X oznacza zmienną. Sprzeczność osiągnie się natychmiast, gdy przyjętym podstawieniem jest X/jan (tj. zamiast X podstawi się jan). A zatem odpowiedź na powyższe pytanie brzmi, że ojcem Piotra jest Jan. Jest to jedyne rozwiązanie.

3. Jakie są imiona wszystkich ojców i dzieci?
To pytanie ma postać:
← ojciec (X,Y)
co jest równoważne **∼ojciec** (X,Y).

Programowaniu w logice jest poświęcona książka [7], a bardziej teoretyczne podejście jest zawarte w [17].

PROGRAMOWANIE W LOGICE A BAZY DANYCH

Relacyjny model danych oraz relacyjne języki zapytań stanowią podzbiór logiki pierwszego rzędu. Charakterystyczną cechą baz danych jest to, że zawierają one:
— duży zbiór nie zinterpretowanych danych, przechowywanych w rekordach i plikach wraz ze strukturami pomocniczymi, np. indeksami,
— schemat bazy danych (na ogół zajmujący znacznie mniej pamięci), reprezentujący pewną wiedzę o modelowanym świecie (tzw. metadane).

Wyodrębnienie schematu jest podyktowane troską o niezależność danych od ich opisu oraz o efektywność wyszukiwania. Schemat zawiera pewien rodzaj reguł dedukcyjnych, zwanych perspektywami (ang. views) [2], które ułatwiają użytkownikom interpretowanie danych i wyprowadzanie z nich nowych informacji. Struktury danych i więzy integralnościowe schematu pojęciowego zapewniają właściwe sterowanie operacjami w bazie danych. Każda z funkcji systemu zarządzania bazami danych jest opisywana innym językiem (języki definicji danych, języki zapytań, itd.). Zaimplementowanie ogólnych reguł dedukcyjnych oraz sprawdzanie więzów integralnościowych wymaga pisania specjalnych programów w macierzystym języku programowania systemu, ponieważ aktualnie stosowane języki zapytań (języki manipulacji danymi) mają zbyt ograniczone możliwości.

W systemach baz danych zakłada się, że rozmiary schematu są niewielkie — niewielka liczba metadanych umożliwia zarządzanie dużymi zbiorami danych. Coraz częściej we współczesnych zastosowaniach potrzebna jest duża liczba metadanych. Coraz częściej też mechanizmy stosowane w tradycyjnych systemach baz danych okazują się niewystarczające. Natomiast, używając logiki pierwszego rzędu można dane, reguły dedukcyjne, więzy integralnościowe i programy działające na bazie danych wyrazić precyzyjnie i jednolicie za pomocą tego samego języka. Co więcej, jest to język deklaratywny, tzn. można określić w nim cel bez podawania sposobu jego osiągnięcia. Zanika formalny podział na schemat, dane i programy. Dużą część danych można opisać za pomocą ogólnych reguł. Na przykład, przedstawioną powyżej genealogiczną bazę danych można uzupełnić o regułę:

brat (X, Y) ← **ojciec** (Z, X), **ojciec** (Z,Y), $X \neq Y$ (4)

stwierdzającą, że X jest bratem Y, jeśli X i Y mają wspólnego ojca. Chcąc się dowiedzieć, czyim bratem jest Piotr, należy zadać pytanie:

← brat (piotr,X)

Odpowiedzią jest ewa, mimo iż w bazie danych nie występuje jawnie informacja:

brat (piotr,ewa)

Tego rodzaju reguły stanowią uogólnienie perspektyw z modelu relacyjnego, gdyż można w nich korzystać z rekurencji, np. w regule:

przodek (X,Y) ← **ojciec** (X,Y)
przodek (X,Y) ← **ojciec** (X,Z), **przodek** (Z,Y)

oraz posługiwać się obiektami o złożonej strukturze, np. w regule znajdowania mieszkańców Opola o nazwisku identycznym z nazwą ulicy, na której mieszkają:

imiennik (X) ← **opolanin** (X, **adres** (X,Nr,Kod))

Za pomocą reguł dedukcyjnych można zapisywać nie tylko perspektywy, lecz także więzy integralnościowe, np. fakt,

[2] W terminologii ANSI/SPARC odpowiada im pojęcie schematu zewnętrznego (ang. external schema) — przyp. aut.

że każdy człowiek ma tylko jednego ojca, można wyrazić za pomocą reguły:

← **ojciec** (Y, X), **ojciec** (Z, X), $Z \neq Y$

Reguły dedukcyjne każdego rodzaju mogą być również stosowane do optymalizacji wykonania zapytań. Ten rodzaj optymalizacji, zwany czasami optymalizacją semantyczną [8, 13], jest trudny do przeprowadzenia w tradycyjnych systemach baz danych.

Język zawierający rekurencję ma większą siłę ekspresji niż tradycyjne języki zapytań stosowane w systemach baz danych. Powstaje jednak nie rozwiązany jak dotąd problem efektywnego wykonania zapytań rekurencyjnych. Propozycje rozwiązań dla różnych klas zapytań rekurencyjnych można znaleźć np. w [6, 22, 9], a przegląd dotychczasowych wyników dokonano w [1]. Posługując się logiką pierwszego rzędu można również znacznie łatwiej badać, a następnie — używając języka programowania w logice — implementować bazy danych z niepełną informacją lub wartościami nijakimi (ang. null values).

Użycie w bazie danych ogólnych reguł zamiast pojedynczych danych znacznie ułatwia wprowadzanie i aktualizację informacji oraz oszczędza miejsce w pamięci. Bazy danych zawierające takie reguły nazywa się często dedukcyjnymi bazami danych. Można w nich uzyskiwać nowe informacje z danych jawnie zapisanych w pamięci. Publikacja [5] stanowi doskonały przegląd tematyki związanej z zastosowaniem logiki w bazach danych i dedukcyjnymi bazami danych, oraz zawiera bogatą bibliografię. Teoretyczne podstawy takich systemów przedstawiono w [18].

Potrzeba rozszerzenia możliwości systemów baz danych przestaje już budzić wątpliwości zarówno wśród ich twórców, jak i użytkowników. Dyskutowane są rozmaite sposoby realizacji takiego ulepszonego systemu. Jedno z badanych podejść polega na wbudowywaniu nowych mechanizmów w już istniejące systemy. Nie omawiając tutaj tego podejścia, warto wspomnieć jedynie, że powstały propozycje rozszerzenia algebry relacji [23], podejmowane są próby wbudowania mechanizmów wnioskowania w system relacyjny INGRES [10, 20], zaproponowano nową operację umożliwiającą stosowanie rekurencji [11], a także opracowano koncepcję systemu, który spełniałby wszystkie wymagania wymienione na wstępie [4].

Dalsza część artykułu jest poświęcona sposobom rozszerzenia możliwości systemu bazy danych przez zastosowanie języka programowania w logice — Prologu.

WPROWADZENIE DO PROLOGU

Prolog (franc. **pro**grammation en **log**ique) został opracowany w 1972 roku w Marsylii przez zespół pod kierunkiem A. Colmerauera. Jest oparty na koncepcji programowania w logice i użyciu metody rezolucji. Do chwili obecnej powstało wiele różnych wersji i dialektów Prologu. Opracowano również inne języki programowania w logice. Chociaż Prolog jest językiem dosyć rozpowszechnionym i ma sporo zastosowań, nie można powiedzieć, że już osiągnął swój kształt ostateczny. Dokonuje się w nim coraz to nowych zmian i udoskonaleń, mających zwiększyć z jednej strony jego siłę wyrazu, a z drugiej — efektywność. Jedną z przyczyn zmian w Prologu są wnioski i potrzeby wynikłe z prób stosowania go w bazach danych. Stosowana dalej składnia Prologu jest oparta (za [16]) na najbardziej popularnej jego wersji, znanej jako Prolog-10. Zaimplementował ją D. H. D. Warren dla komputera DEC-10. W artykule przedstawiono jedynie najważniejsze cechy Prologu. Pełny opis języka można znaleźć w [15] (omówiono tam oryginalną wersję marsylską) i w [3]. Materiał dla bardziej zaawansowanych czytelników zawierający niebanalne przykłady i opis implementacji Prologu znajduje się w [16].

W Prologu nie istnieje pojęcie typu danych. Można posługiwać się obiektami prostymi i złożonymi, których opisy są nazywane termami. Obiektom prostym, zazwyczaj utożsamianym ze stałymi, nie przypisuje się żadnej ustalonej interpretacji, o ile nie są użyte jako argumenty standardowych operacji arytmetycznych, porównania lub wejścia-wyjścia. Obiektami złożonymi są drzewa i listy. Termy reprezentujące takie obiekty zapisuje się w postaci nazwy, zwanej funktorem, po której w nawiasie następują termy opisujące składowe. Prolog umożliwia stosowanie dowolnych funktorów jako funktorów pozycyjnych (prefiksowych, infiksowych lub postfiksowych). Funktorem jest na przykład term kropka, który oddziela elementy listy. Listy zapisuje się w postaci [Głowa | Ogon], gdzie Głowa odpowiada pierwszemu elementowi listy, a Ogon — reszcie. Lista pusta jest oznaczana [ ]. Obiekty o nieznanej strukturze są nazywane zmiennymi, a ich nazwy rozpoczynają się od dużej litery. W trakcie obliczeń zmienna może zostać ukonkretniona (związana z termem), w wyniku czego opis reprezentowanego przez nią obiektu staje się bardziej precyzyjny. Term oraz związana z nim zmienna są wzajemnie nieodróżnialne. Taki term może zawierać inne zmienne. Oto kilka przykładów termów wraz z ich interpretacją:
X — wszystkie typy obiektów,

[G | 0] — wszystkie niepuste listy,

zwierzur (X,zoo wrocław,1986) — wszystkie zwierzęta urodzone we wrocławskim zoo w 1986 roku,

[A.x] — wszystkie dwuelementowe listy o drugim elemencie równym x,

[A.A | T] — wszystkie, co najmniej dwuelementowe, listy z identycznymi pierwszymi dwoma elementami.

Program w Prologu jest zbiorem procedur złożonych z klauzul. Klauzulę tworzy nagłówek i, być może pusta, treść. Procedurą nazywa się zbiór klauzul, których nagłówki mają tę samą nazwę i liczbę parametrów. Nagłówek klauzuli jest oddzielany od jej treści symbolem :—, a koniec klauzuli jest oznaczany kropką. Procedurą jest na przykład:

brat (X, Y) :— **ojciec** (Z, X), **ojciec** (Z, Y), $X \neq Y$.

brat (X, Y) :— **matka** (Z, X), **matka** (Z, Y), $X \neq Y$.

Pierwsza z tych klauzul dokładnie odpowiada klauzuli (4). Klauzule o tej samej nazwie i różnej liczbie parametrów należą do różnych procedur, np.:

ojciec (X) :— ojciec (X,Y).

Każdą klauzulę postaci:

p :— q,r,s.

można interpretować na dwa sposoby. Pierwsza interpretacja, zwana deklaratywną lub pragmatyczną, stwierdza, że zachodzi p, jeśli zachodzi q, r i s. Druga, nazywana proceduralną, mówi, że aby obliczyć p należy najpierw obliczyć q, r is (czyli wywołać te procedury). Na przykład procedura:

student (X) :— matura (X,_), pktegz (X,P), P>16.

definiuje studenta jako posiadacza matury, który otrzymał więcej niż 16 punktów egzaminacyjnych (interpretacja deklaratywna). Zakładamy, że drugi parametr procedury matura określa nazwę szkoły średniej. Podkreślenie oznacza zmienną anonimową. Stosuje się ją wówczas, gdy pewne obiekty nie są konieczne w dalszej fazie wykonania procedury i nie ma potrzeby odwoływania się do nich. Można również powiedzieć (interpretacja proceduralna), że aby stwierdzić, czy X jest studentem, należy sprawdzić, czy ukończył szkołę średnią, a następnie obliczyć liczbę otrzymanych przez niego punktów i sprawdzić, czy jest większa od 16. Jeśli do tej procedury dołączy się klauzulę:

student (X) :— matura (X,_), olimpijczyk (X).

to do poprzedniej deklaratywnej interpretacji należy dodać: „lub posiada maturę i jest finalistą olimpiady". Różne klauzule tej samej procedury oznaczają różne możliwe drogi obliczenia.

Jak widać z powyższego przykładu w Prologu przyjęto, że procedury występujące w treści klauzuli są wykonywane w kolejności od lewej do prawej. Natomiast klauzule są wybierane w kolejności ich pojawienia się w programie. Oczywiście, do wykonania wybiera się klauzule, których nagłówek można uzgodnić z wywołaniem procedury. W jednym wywołaniu procedury każda klauzula może być wykonana tylko jeden raz. Gdy pewne wywołanie procedury zawiedzie, tzn. algorytm unifikacji nie znajdzie obiektów spełniających opisywaną przez nią relację (nie znajdzie unifikatora), to wykonywany jest nawrót. Polega on na anulowaniu części uzyskanych ostatnio wyników i odtworzeniu historii obliczeń aż do napotkania procedury mającej jeszcze nieuaktywnione klauzule o nagłówkach uzgadnialnych z jej wywołaniem. Można to prześledzić na przykładzie genealogicznej bazy danych.

Niech program w Prologu składa się z następujących procedur:

ojciec (jan,piotr).
ojciec (piotr,anna).
ojciec (jan,ewa).
mężczyzna (jan).
mężczyzna (piotr).

brat (X, Y) :— **ojciec** (Z, X), **ojciec** (Z, Y), X≠Y,
mężczyzna (X).

brat (X, Y) :— **matka** (Z, X), **matka** (Z, Y), X≠Y,
mężczyzna (X).

W celu znalezienia brata Piotra należy napisać:

:— **brat** (piotr,X), **mężczyzna** (X), write (X).

Najpierw zostanie wywołana pierwsza z klauzul procedury brat. Algorytm unifikacji przypisze parametrom formalnym odpowiednie termy: X/piotr, Y/X. Otrzymuje się wówczas:

:— **ojciec** (Z, piotr), **ojciec** (Z, X), piotr≠X,
mężczyzna piotr, mężczyzna X, write X.

(podkreśleniem wyróżniono aktualnie wywołaną procedurę). Następuje pierwsze ukonkretnienie zmiennej Z, gdyż podstawiając Z/jan można dokonać unifikacji z pierwszą klauzulą procedury **ojciec**. Zatem:

:— **ojciec** (jan, piotr), **ojciec** (jan, X), piotr≠X,
mężczyzna (piotr), **mężczyzna** (X), write X.

Algorytm unifikacji ponownie użyje pierwszej klauzuli procedury **ojciec** (jest to inne wywołanie tej procedury) i zwiąże zmienną X z termem piotr. Stąd otrzymuje się:

:— **ojciec** (jan, piotr), **ojciec** (jan, piotr), piotr≠piotr,
mężczyzna (piotr), **mężczyzna** (piotr), **write** (piotr).

Oczywiście nastąpi nawrót. Zmienna X traci swoją wartość piotr i następuje próba jej ponownego ukonkretnienia na podstawie kolejnych klauzul procedury **ojciec**. Zmienna X jest teraz związana z termem ewa z trzeciej klauzuli procedury **ojciec** (druga klauzula tej procedury ewidentnie nie da się uzgodnić). Jednak po pomyślnym sprawdzeniu, że piotr≠ewa oraz że Piotr jest mężczyzną, zawodzi wywołanie procedury **mężczyzna** (ewa). Musi zatem nastąpić nawrót do wywołania **ojciec** (jan,X) anulujący poprzednie ukonkretnienie zmiennej X. To wywołanie również zawodzi, gdyż nie ma już nie zbadanych klauzul procedury **ojciec**. Następuje kolejny nawrót do poprzedniego wywołania tej procedury i próba znalezienia innego ojca Piotra czyli innego ukonkretnienia zmiennej Z w **ojciec** (Z,piotr). Oczywiście nie ma takiego, a zatem zawodzi pierwsza klauzula procedury **brat**. Możliwe jest jednak uzgodnienie z wywołaniem **brat** (piotr,X) nagłówka drugiej klauzuli. Mamy wówczas:

:— **matka** (Z, piotr), **matka** (Z, X), piotr≠X,
mężczyzna (piotr), **mężczyzna** (X), write X.

Algorytm unifikacji nie znajduje procedury **matka**, co w większości implementacji Prologu jest traktowane jako niepowodzenie. Próba nawrotu również kończy się niepowodzeniem, ponieważ nie ma już nie zbadanych klauzul procedury brat. Zatem w odpowiedzi otrzymuje się informację „nie ma".

Unieważnienie części dotychczasowych wyników podczas nawrotu nie odnosi się do procedur standardowych, powodujących efekty uboczne, takie jak wypisanie na ekran lub dynamiczne dodawanie i usuwanie klauzuli. Wyników działania takich procedur nie można anulować.

W implementacji Prologu przyjęto na ogół, że po uzyskaniu odpowiedzi użytkownik może zażądać następnej, wprowadzając znak „;". Jeśli jednak jego zamiarem jest znalezienie wszystkich odpowiedzi, np. wszystkich dzieci Jana, to może napisać:

:— **ojciec** (jan,X), write (X), **fail**,

Słowo **fail** jest standardowo przyjętą nazwą procedury powodującej niepowodzenie. Zatem zawsze po znalezieniu odpowiedzi (tj. po ukonkretnieniu X) nastąpi nawrót, a gdy wywołanie procedury **ojciec** zawiedzie (X nie da się ukonkretnić), użytkownik otrzymuje informację „nie ma więcej".

Łatwo zauważyć, że automatyczne nawracanie po każdym niepowodzeniu może powodować znaczne marnotrawienie czasu i pamięci. Widać to w powyższym przykładzie, gdy próbuje się znaleźć drugiego ojca Piotra. Aby dać programiście możliwość pewnego sterowania nawrotami wprowadzono operator odcięcia oznaczony „!". Jego wystąpienie uniemożliwia dokonanie nawrotu do procedur występujących przed nim i tym samym oznacza ostateczną akceptację wszystkich osiągniętych wcześniej wyników. Na przykład, procedurę sprawdzającą, czy dwa zbiory są rozłączne można zapisać jako:

rozłączne (Zb1,Zb2) :— elzb (El,Zb1), elzb (El,Zb2), !, **fail**.
rozłączne (Zb1,Zb2).

Zakłada się, że procedura elzb (X,Y) sprawdza, czy X jest elementem zbioru Y.

Widać, że jeśli istnieje element należący do obu zbiorów Zb1 i Zb2, to nie ma potrzeby kontynuowania obliczeń. Odcięcie zapobiegnie nawrotowi i procedura zawiedzie. Gdyby odcięcia nie było, to wskutek kolejnych nawrotów niepotrzebnie byłyby sprawdzane wszystkie elementy zbioru Zb1.

Jeśli nie istnieje element należący do obu zbiorów, to nie dojdzie do wywołania odcięcia i po nawrocie nastąpi wywołanie drugiej, zawsze spełnionej klauzuli.

Odcięcie psuje jednak deklaratywną interpretację procedury i znacznie zmniejsza jej czytelność. W wielu wypadkach, zamiast odcięcia wystarczy zastosować standardową procedurę **not**, która kończy się powodzeniem, gdy zawodzi próba wywołania procedury będącej jej parametrem. Na przykład mając daną procedurę **mającząśćwspólną**, sprawdzającą, czy dwa zbiory mają część wspólną, procedurę **rozłączne** można zdefiniować jako:

rozłączne (Zb1,Zb2) :— **not** (mającząśćwspólną (Zb1,Zb2)).

Jedną z charakterystycznych cech Prologu jest możliwość stosowania tej samej procedury w różnych celach. Typowym przykładem jest procedura append:

append ([], L, L).
append ([Pocz | Kon], L, [Pocz | Kon 1]) :— append (Kon, L, Kon1).

Definiuje ona, w jaki sposób lista, będąca trzecim parametrem, powstaje z połączenia list opisanych dwoma pierwszymi parametrami. Lista L powstaje z połączenia L z listą pustą, a lista [Pocz | Kon 1] powstaje z połączenia list L i [Pocz | Kon], jeśli Kon 1 jest złączeniem list L i Kon. W zależności od sposobu wywołania append można listy łączyć lub rozdzielać. Na przykład:

:— append ([a.b], [c], L), write (L).
da w wyniku listę [a.b.c], a wywołanie:
:— append (P, K, [a.b.c]), write (P), write ('&'),
write (K), nl, fail.
(nl — przejście do nowej linii)

da w wyniku:

[ ] & [a.b.c]
[a] & [b.c]
[a.b] & [c]
[a.b.c] & [ ]

LITERATURA

[1] Bancilhon F., Ramakrishnan R.: An Amateur's Introduction to Recursive Query Processing Strategies. Proc. of ACM SIGMOD Conf. on Management of Data, Washington, 1986
[2] Brodie M. L. (ed.): Proc. of Workshop on Large Scale Knowledge Base and Reasoning Systems, Islamorada, 1985
[3] Cloksin W. F., Mellish C. S.: Programming in Prolog. Springer Verlag, Berlin, 1981
[4] Dayal U., Smith J. M.: PROBE — A Knowledge-Oriented Database Management System [2]
[5] Gallaire H., Minker J., Nicolas J.-M.: Logic and Databases — A Deductive Approach. Computing Surveys, Vol. 16, No. 2, 1984
[6] Henshen L. J., Naqvi E. A.: On Compiling Queries in Recursive First-Order Databases. Journal of ACM, Vol. 31, No. 1, 1984
[7] Hogger C. J.: Introduction to Logic Programming. Academic Press, New York, 1984
[8] Hammer M., Zdonik S.: Knowledge-bases Query Processing. Proc. of 6th Conf. of VLDB, Montreal, 1980
[9] Ioannidis Y. E.: A Time Bound cn The Materialization of Some Recursively Defined Views. Proc. of 11th Conf. of VLDB, Stockholm, 1985

dokończenie na str. 11

JAN BIELECKI
Instytut Informatyki
Politechnika Warszawska

Podstawy grafiki w języku Turbo Pascal (1)

Przegląd środków graficznych

W języku Turbo Pascal dla mikrokomputera IBM PC współpraca z monitorem ekranowym może odbywać się w jednym z czterech trybów tekstowych albo w jednym z trzech trybów graficznych. W trybie tekstowym ekran monitora jest podzielony na 25 wierszy, a w każdym wierszu można wyprowadzić do 40 albo do 80 znaków. Wyprowadzane znaki mogą być czarno-białe albo kolorowe. W trybach graficznych ekran monitora stanowi raster składający się z 200 linii. W każdej linii można wyświetlić 320 albo 640 pikseli. W zależności od użytego trybu i monitora wyświetlane punkty mogą być czarno-białe albo kolorowe.

W zwykłych warunkach wykonywanie operacji ekranowych dotyczy całego ekranu. Istnieje jednak możliwość zdefiniowania prostokątnego fragmentu ekranu jako okienka i wykonywanie operacji ekranowych tak, jakby to okienko zajmowało cały dostępny ekran. W każdej chwili może być zdefiniowane tylko jedno okienko. W trybach tekstowych będzie ono nazywane okienkiem tekstowym, a w trybach graficznych — okienkiem graficznym. Rozszerzenie języka o grafikę żółwia [1]) umożliwia ponadto zdefiniowanie okienka żółwia. Jeśli nie zdefiniowano okienka żółwia, to przez domniemanie przyjmuje się, że jest ono identyczne z okienkiem domniemanym, którym jest cały ekran.

Bezpośrednio po jawnym albo domniemanym zdefiniowaniu okienka wszystkie operacje dotyczą tylko tego okienka. Oznacza to, że w wypadku okienka tekstowego będzie zachodziło pionowe przeglądanie tekstu nie mieszczącego się w okienku (ang. scrolling), a w wypadku okienka graficznego — obcinanie (ang. clipping). Bliższe informacje na ten temat zostaną podane w punkcie poświęconym zarządzaniu okienkami.

TRYBY TEKSTOWE

Ustanowienie trybu tekstowego odbywa się za pomocą procedury TextMode. Argumentami procedury TextMode są zazwyczaj predefiniowane literały typu integer: BW40, C40, BW80 i C80 reprezentujące dane o wartościach 0, 1, 2 i 3. Wykonanie procedury TextMode bez argumentu powoduje ponowne ustanowienie poprzedniego trybu tekstowego.

Wykonanie procedury TextMode(BW40) powoduje ustanowienie trybu tekstowego z wyświetlaniem znaków czarno-białych (ang. black and white), do 40 znaków w wierszu.

Wykonanie procedury TextMode(C40) powoduje ustanowienie trybu tekstowego z wyświetlaniem znaków kolorowych (ang. color), do 40 znaków w wierszu.

Wykonanie procedury TextMode(BW80) powoduje ustanowienie trybu tekstowego z wyświetlaniem znaków czarno-białych (black and white), do 80 znaków w wierszu.

Wykonanie procedury TextMode(C80) powoduje ustanowienie trybu tekstowego z wyświetlaniem znaków kolorowych, do 80 znaków w wierszu.

Każde ustanowienie trybu tekstowego powoduje wyczyszczenie ekranu, tj. zapełnienie go spacjami o kolorze tła, a ponadto przemieszczenie kursora do lewego górnego narożnika ekranu.

Przykład 1. Wyświetlanie znaków w trybie tekstowym

```
program Ewa;
begin
  TextMode;
  Write('Ewa');
  repeat until KeyPressed;
  TexMode(BW40)
end.
```

- Bezpośrednio po aktywowaniu programu, ale przed wykonaniem procedury TextMode, ekran nie musi być wyczyszczony.
- Wykonanie procedury TextMode bez argumentu spowoduje wyczyszczenie ekranu i ustanowienie trybu tekstowego domniemanego C80.
- Po wyczyszczeniu ekranu kursor znajduje się w jego lewym górnym narożniku.
- Wykonanie instrukcji Write powoduje wyprowadzenie w pierwszym wierszu ekranu napisu Ewa.
- Po naciśnięciu dowolnego znaku klawiatury nastąpi ustanowienie trybu tekstowego BW40. Spowoduje to wyczyszczenie ekranu.
- Po zakończeniu wykonywania programu, wyprowadzanie na ekran pozostaje w trybie BW40.

W trybach tekstowych z użyciem kolorów, C40 i C80, każdy znak może być wyświetlany w jednym z 16 kolorów na dowolnym tle wybranym spośród 8 kolorów. Na kolory można powoływać się za pomocą literałów typu integer, takich jak LightBlue, Yellow itp. Pełne zestawienie tych literałów zamieszczono w tabeli 1. Znaki mogą być wyświetlane zarówno w kolorach ciemnych jak i jasnych. Tło mogą stanowić jedynie kolory ciemne. Posługując się wyświetlaniem w kolorach należy wziąć pod uwagę, że w niektórych monitorach nie ma możliwości odróżnienia kolorów ciemnych od jasnych. W takich wypadkach kolory ciemne są wyświetlane tak, jak ich jasne odpowiedniki. W szczególności, kolor żółty (literał Yellow) może być wyświetlany jako brązowy (Brown), a kolor biały (White) może być wyświetlany jako jasnoszary (LightGray).

Określenie koloru wyświetlanych znaków odbywa się za pomocą procedury TextColor, a określenie koloru tła za pomocą procedury TextBackground. Jeśli do liczby określającej kolor zostanie dodana liczba 16, zazwyczaj reprezentowana za pomocą predefiniowanej nazwy literału — Blink, to wyświetlany znak będzie migotać.

Określenie koloru znaków wyprowadzanych na ekran odbywa się za pomocą procedury TextColor. W ogólnym wypadku procedura ta ma postać:

TextColor(Hue)

W zapisie tym Hue jest wyrażeniem typu integer reprezentującym daną o wartości z przedziału 0...31. Jeśli dana reprezentowana przez Hue ma wartość większą od 15, to argument Hue jest traktowany tak, jakby miał wartość Hue-15, a wyświetlane znaki są pobudzane do migotania. Kolor wyświetlanych znaków jest określony przez wyrażenie Hue mod 16 i wynika z zestawienia przytoczonego w tabeli 1.

[1]) Zdaniem lingwistów, których prosiłem o opinię, termin „turtle graphics" powinien być tłumaczony na: „grafika żółwiowa". Niepoprawne tłumaczenie „grafika żółwia" pozostawiam na życzenie Redakcji. (JB)

Tabela 1. Mnemoniczne nazwy literałów oznaczających kolory i migotanie

Literał	Nazwa (0—7 kolory ciemne, 8—15 — jasne)
0	Black (czarny)
1	Blue (niebieski)
2	Green (zielony)
3	Cyan (turkusowy)
4	Red (czerwony)
5	Magenta (karmazynowy)
6	Brown (brązowy)
7	DarkGrey (ciemnoszary)
8	LightGrey (jasnoszary)
9	LightBlue (jasnoniebieski)
10	LightGreen (jasnozielony)
11	LightCyan (jasnoturkusowy)
12	LightRed (jasnoczerwony)
13	Light Magenta (jasnokarmazynowy)
14	Yellow (żółty)
15	White (biały)
16	Blink (migotanie)

Przykład 2. Wyświetlanie znaków w kolorach

```
program EwaIzaJan;
begin
  TextMode(C40);
  Write('Ewa');
  TextColor(Red);
  Write('Iza');
  TextColor(Green + Blink);
  Write('Jan');
  repeat until KeyPressed;
  TextMode(BW80)
end.
```

● Napis Ewa zostanie wyświetlony w kolorze domniemanym — żółtym i na tle domniemanym — czarnym.

● Napis Iza zostanie wyświetlony na tle domniemanym i w kolorze czerwonym.

● Napis Jan zostanie wyświetlony na tle domniemanym i w kolorze zielonym. Litery tego napisu będą migotać.

● Po naciśnięciu dowolnego znaku klawiatury nastąpi przywrócenie trybu tekstowego BW80 i zakończenie wykonywania programu.

Określenie koloru tła odbywa się za pomocą procedury TextBackground. W ogólnym wypadku procedura ta ma postać:

TextBackground(Hue)

W zapisie tym Hue jest wyrażeniem typu integer, reprezentującym daną o wartości z przedziału 0..7. Każdej z wartości tej danej odpowiada jeden z kolorów ciemnych wyszczególnionych w pierwszej części tabeli 1. Po wykonaniu procedury TextBackground wszystkie następnie wyprowadzane znaki są wyświetlane na tle o numerze Hue.

Przykład 3. Wyświetlanie znaków na obranym tle

```
program Iza;
begin
  TextMode(C40)
  TextBackground(Green);
  TextColor(Brown);
  Write('Jan');
  repeat until KeyPressed
end.
```

● Napis Iza zostanie wyprowadzony w kolorze brązowym (Brown) na zielonym tle (Green).

● Po naciśnięciu dowolnego klawisza klawiatury nastąpi zakończenie wykonywania programu.

Każde wyprowadzenie znaków na ekran powoduje zmianę pozycji kursora. W chwili ustanowienia trybu graficznego kursor znajduje się w lewym górnym narożniku ekranu. Ustalenie pozycji kursora może być wykonane za pomocą funkcji WhereX i WhereY. Obie te funkcje są bezargumentowe. Wywołanie funkcji ma postać:

WhereX
oraz
WhereY

Każda z nich może wystąpić jako element wyrażenia.

Przykład 4. Lokalizowanie pozycji kursora

```
program Cursor;
begin
  TextMode;
  Write (WhereX,WhereY :2);
  GotoXY (20,30);
  Write (WhereX + WhereY) ;
  repeat until KeyPressed
end.
```

● Wykonanie programu powoduje wyprowadzenie liczb 1, 1 i 50.

● Bezpośrednio po ustanowieniu trybu tekstowego (procedura TextMode) kursor znajduje się w punkcie o współrzędnych (1,1).

● Bezpośrednio po wykonaniu procedury GotoXY kursor znajduje się w punkcie o współrzędnych (20,30).

● Zakończenie wykonywania programu następuje po naciśnięciu dowolnego klawisza klawiatury.

TRYBY GRAFICZNE

Ustanowienie trybu graficznego odbywa się za pomocą procedur GraphColorMode, GraphMode i HiRes. Dwie pierwsze z tych procedur powodują ustanowienie trybu graficznego średniej rozdzielczości, 200 × 320 pikseli, a trzeci powoduje ustanowienie trybu graficznego wysokiej rozdzielczości, 200 × 640 pikseli. Wywołanie każdej z procedur powoduje wyczyszczenie ekranu, tj. zapełnienie go kolorem tła. Przed zakończeniem wykonywania programu, w którym posługiwano się trybem graficznym, należy zapewnić ustanowienie jednego z trybów tekstowych.

Wyświetlanie w trybie GraphColorMode odbywa się w kolorze, a obraz wyświetlany w trybie GraphMode jest czarno-biały. Tym niemniej — na monitorze RGB — nawet w trybie GraphMode istnieje możliwość wyboru barwy. Obraz wyświetlany w trybie wysokiej rozdzielczości jest także czarno-biały, jednak i w tym trybie, za pomocą procedury HiResColor, można określić kolor wykreślanych obiektów. Tło natomiast może być tylko czarne. Jeśli procedura HiResColor nie zostanie wywołana, to wyświetlanie będzie odbywać się w domniemanym kolorze białym.

Kolory pikseli mogą być wybierane z palety barw dobieranej za pomocą procedury Palette. W ogólnym wypadku wywołania procedury Palette ma postać:

Palette(Num)

W zapisie tym Num jest wyrażeniem typu integer reprezentującym daną, której wartość określa numer palety barw. Dopuszczalne palety barw dla monitora kolorowego przedstawiono w tabeli 2, a dla monitora RGB w tabeli 3.

Tabela 2. Palety barw monitora kolorowego

Numer koloru	Paleta 0	Paleta 1
0	Tło	Tło
1	Zielony	Turkusowy
2	Czerwony	Karmazynowy
3	Brązowy	Jasnoszary

Tabela 3. Palety barw monitora RGB

Numer koloru	Paleta 0	Paleta 1
0	Tło	Tło
1	Niebieski	Jasnoniebieski
2	Czerwony	Jasnoczerwony
3	Jasnoszary	Biały

Jak wynika z tabel, wykreślanie w kolorze nr 0 jest niezależne od obranej palety i odbywa się zawsze w kolorze tła. W trybach graficznych średniej rozdzielczości kolor tła może być określony za pomocą procedury GraphBackground. W trybie wysokiej rozdzielczości nie istnieje pojęcie palety, a tło jest zawsze czarne. W trybie tym możliwe jest jedynie ustalenie koloru wyświetlanych pikseli za pomocą procedury HiResColor.

Wywołanie procedury GraphBackground ma w ogólnym przypadku postać:

GraphBackground(Hue)

W zapisie tym Hue jest wyrażeniem typu integer, reprezentującym daną o wartości z przedziału 0..15. Wyrażenie to ma zazwyczaj postać nazwy literału. Związek wartości literału z jego nazwą wynika z tabeli 1. W trybie średniej rozdzielczości każde wykonanie procedury GraphBackground powoduje natychmiastową zmianę koloru tła. W trybie wysokiej rozdzielczości wywołanie tej procedury nie powoduje żadnych skutków.

Wywołanie procedury HiResColor ma w ogólnym wypadku postać:

HiResColor(Hue)

W zapisie tym Hue jest wyrażeniem typu integer o wartości z przedziału 0..15. Związek wartości literału z jego nazwą wynika z tabeli 1. Każdorazowe wywołanie procedury HiResColor z odmienną od dotychczasowej wartością argumentu powoduje natychmiastową zmianę koloru wszystkich punktów wyświetlanych w trybie HiRes. W pozostałych trybach wykonanie omawianej procedury nie wywołuje żadnych skutków.

Ustanowienie trybu graficznego nie wyklucza możliwości wyprowadzania na ekran tekstu. Znaki tekstu, wyprowadzane za pomocą procedury Write, są umieszczane na takich samych pozycjach jak w trybach tekstowych. W odróżnieniu od sprzętowego wyświetlania w trybach tekstowych, w trybach graficznych znaki tekstu są wykreślane punkt po punkcie w kolorze nr 3 wybranej palety.

Wykonanie bezargumentowej procedury GraphColorMode powoduje ustanowienie trybu graficznego kolorowego o rozdzielczości 200 × 320 pikseli. W tym trybie współrzędne poziome pikseli przybierają wartości z przedziału 0..319. Lewy górny piksel ekranu ma współrzędne (0,0). Po wybraniu palety i koloru tła można uzyskać wyświetlanie punktów w dowolnym z 4 kolorów palety.

Przykład 5. Wyświetlanie znaków w trybie GraphColorMode

```
program Ewa;
begin
  GraphColorMode;
  Palette(1) ;
  GraphBackground(Blue);
  Write('Ewa') ;
  repeat until KeyPressed
end.
```

● Wykonanie programu powoduje wyświetlenie w trybie graficznym kolorowym napisu Ewa.

● Cały ekran zostaje wypełniony kolorem tła — niebieskim, a napis zostaje wyświetlony w kolorze nr 3 wybranej palety barw, tj. jasnoszarym.

● Gdyby z programu usunięto wywołania procedur Palette i GraphBackground, to domniemaną byłaby paleta nr 0, a domniemanym kolorem tła — czarny. W takim wypadku, wspomniany napis zostałby wyświetlony w kolorze nr 3 domniemanej palety, tj. brązowym.

Wykonanie bezargumentowej procedury GraphMode powoduje ustanowienie trybu graficznego czarno-białego o rozdzielczości 200 × 320 pikseli. W tym trybie współrzędne pionowe pikseli przybierają wartości z przedziału 0..199, a współrzędne poziome przybierają wartości z przedziału 0..319. Lewy górny piksel ekranu ma współrzędne (0,0). Na monitorze RGB, takim jak np. monitor kolorowy firmy IBM, jest możliwe wyświetlanie w kolorze. Po wybraniu jednej z 2 palet wyszczególnionych w tabeli 3 (procedura Palette) oraz wybraniu koloru tła (procedura GraphBackground) można uzyskać wyświetlanie punktów o dowolnym z 4 kolorów tej palety.

Przykład 6. Wyświetlanie znaków w trybie GraphMode

```
program Ewa;
begin
  GraphMode;
  Palette(1) ;
  GraphBackground(Green);
  Write('Ewa') ;
  repeat until KeyPressed;
  TextMode
end.
```

● Na monitorze RGB napis Ewa zostanie wyświetlony w kolorze nr 3 palety nr 1 — tj. białym, na zielonym tle.
● Gdyby z programu usunięto wywołania procedur Palette i GraphBackground, to napis Ewa zostałby wyświetlony w kolorze brązowym na czarnym tle.

Wykonanie bezargumentowej procedury HiRes powoduje ustanowienie trybu graficznego czarno-białego o rozdzielczości 200 × 640 pikseli. W tym trybie, nazywanym także trybem wysokiej rozdzielczości, współrzędne pionowe pikseli przybierają wartości z przedziału 0..199, a współrzędne poziomie przybierają wartości z przedziału 0..639. Punkty są wyświetlane na czarnym tle, w kolorze wybranym za pomocą procedury HiResColor.

Przykład 7. Wyświetlanie znaków w trybie HiRes

```
program Ewa;
begin
  HiRes;
  HiResColor(Black);
  Write('Ewa') ;
  repeat until KeyPressed;
  HiResColor(Red);
  Delay(1000);
  TextMode
end.
```

● Wykonanie instrukcji Write powoduje wyprowadzenie napisu Ewa.
● Napis ten jest niewidoczny, ponieważ wyświetlanie odbywa się w kolorze tła — czarnym.
● Po wprowadzeniu z klawiatury dowolnego znaku, na ekranie ujawnia się czerwony napis Ewa i po 10 s znika.

Wszystkie wymienione dotąd procedury należą do podstawowego zestawu podprogramów graficznych i nie wymagają deklarowania. Do zestawu tego należą także procedury do definiowania okienek: Window i GraphWindow oraz procedury do wyświetlania punktu i odcinka: Plot i Draw. Wszystkie pozostałe podprogramy graficzne, w tym liczne podprogramy do realizowania grafiki żółwia, znajdują się w pliku Graph.bin i wymagają włączenia do programu deklaracji zawartych w pliku Graph.p. Odbywa się to za pomocą dyrektywy włączenia, mającej postać komentarza zawierającego znaki $i oraz nazwę pliku.

Przykład 8. Włączenie deklaracji z pliku Graph.p

```
program Any;
{$i Graph.p}
...
begin
  ...
end.
```

ERRATA. W artykule dr. Jana Madeya „Problematyka systemów operacyjnych..." w numerze 6, 1987, str. 2, prawa szpalta, 36 wiersz od końca powinno być:
opis zasobów, a nie opis zespołów — jak mylnie wydrukowano. Redakcja przeprasza Autora i Czytelników za zniekształcenie tekstu.

Ogłoszenia ● Ogłoszenia ● Ogłoszenia ● Ogłoszenia

Spółdzielnia Rzemieślnicza „Elmech", ul. Dobra 56, 00-312 Warszawa, oferuje owijarki elektryczne (pistoletowe) do połączeń „wire-wrap" przystosowane do drutu ϕ 0,20—0,35 mm. Informacje — telefon 22-94-46.

EO/736/87

Ogłoszenia ● Ogłoszenia ● Ogłoszenia ● Ogłoszenia

MARY M. SHAW
Software Engineering Institute
Pittsburgh, stan Pensylwania
Uniwersytet Carnegie-Mellon
USA

Poza programowanie wielkoskalowe — kolejne wyzwania dla inżynierii oprogramowania (2)

W drugiej części artykułu, na tle dotychczasowego rozwoju inżynierii oprogramowania, przedstawiono dwa zupełnie nowe zespoły zagadnień, które będą wymagały rozwiązania w następnym dziesięcioleciu.

ROZSZERZENIE ZAKRESU TECHNIKI OBLICZENIOWEJ

Istota techniki obliczeniowej, a więc także inżynierii oprogramowania, zmienia się gwałtownie. Aby inżynieria oprogramowana była przygotowana do rozwiązywania problemów lat dziewięćdziesiątych, trzeba już teraz rozpoznawać czynniki, które wpływają na kształt tej dziedziny, i przewidzieć rolę, jaką komputery i oprogramowanie będą odgrywać w przyszłości. W tym punkcie przedstawiono pewne nowe tendencje, które mogą wystąpić w następnym dziesięcioleciu, a także opisano niektóre nowe i przewidywane zjawiska i zagadnienia.

Komputery stają się coraz mniejsze i tańsze, dlatego zwiększa się ich upowszechnienie we wszystkich dziedzinach życia. Do najważniejszych tendencji rozwojowych można zaliczyć:

- zmniejszanie się kosztów sprzętu,
- zwiększenie udziału oprogramowania w całkowitych kosztach systemów obliczeniowych,
- wzrost oczekiwań dotyczących mocy obliczeniowej i niezawodności systemów aplikacyjnych,
- poszerzenie się zakresu zastosowań, szczególnie w kierunkach, gdzie istotne jest życie ludzkie,
- zwiększenie roli oprogramowania jako składowej niejednorodnych systemów zintegrowanych,
- intensywny rozwój systemów rozproszonych i sieci komputerowych,
- zwiększenie roli komputerów jako podstawy usług informacyjnych i druku elektronicznego,
- polepszenie jakości sprzężenia z człowiekiem (komunikacja głosowa, grafika o wysokiej rozdzielczości),
- zwiększenie dostępności mocy obliczeniowej, szczególnie w domach,
- zwiększenie częstotliwości kontaktów z komputerem osób bez przygotowania, zarówno w domu jak i w pracy,
- coraz szersze opieranie się na komputerach w wykonywaniu codziennych, rutynowych czynności,
- ciągły i pogłębiający się brak wykwalifikowanych specjalistów,
- wzrost znaczenia systemów inteligentnych, podejmujących złożone decyzje,
- wzrost zainteresowań prawnymi i ekonomicznymi aspektami obrotu oprogramowaniem.

Na podstawie tych tendencji można przewidywać przyszłość niektórych kierunków techniki obliczeniowej.

Systemy powszechnego użytku. Komputery staną się niezwykle rozpowszechnione jako maszyny uniwersalne w domach i biurach, a także jako maszyny specjalizowane np. w ochronie środowiska czy systemach zabezpieczeń. Zatem większość użytkowników tych komputerów (najczęściej bez przygotowania, a z pewnością nieprogramiści) będzie korzystać z gotowych programów, nie będąc ich twórcami.

Usługi informacyjne. O komputerach zacznie się myśleć przede wszystkim jako o narzędziach do komunikacji i dostępu do informacji, a nie jako o maszynach służących do obliczeń. Nowoczesne sieci komputerowe staną się środkiem w większym zakresie udostępniającym publiczne bazy danych, w sensie biernym (katalogi, biblioteki, czasopisma, wydawnictwa nieperiodyczne) jak i czynnym (informatory, poczta elektroniczna, rozrywki indywidualne). Większego znaczenia nabiorą zastosowania do sterowania w czasie rzeczywistym.

Szeroki zakres zastosowań. Nadal będzie poszerzać się zakres zastosowań komputerów, a szczególnie wzrośnie liczba zastosowań, w których od niezawodności oprogramowania będzie zależeć ludzkie życie. Ze względu na użytkowanie głównie przez nieprogramistów, większość oprogramowania będzie wymagała uproszczonej obsługi z minimalną bądź w ogóle zbędną znajomością zasad programowania. Zaistnieje uzasadniona ekonomicznie potrzeba wytwarzania systemów uniwersalnych, które będzie można dostosowywać do indywidualnych, być może bardzo szczególnych wymagań.

Przemiany w miejscu pracy. Systemy rozproszone i sieci komputerowe ułatwią rozproszenie stanowisk pracy, lecz należy wątpić, że regułą dla pracowników biurowych stanie się praca w domu zamiast w biurze. Łączność elektroniczna przyspieszy komunikację, jednak komputery nie zastąpią udziału człowieka w podejmowaniu decyzji. Skomputeryzowane stanowiska robocze zmienią istotę pracy, która polega na przepływie dokumentów, a robotyzacja spowoduje radykalne zmiany w produkcji.

Wielkie zestawy komputerowe. Niektóre sieci komputerowe i duże komputery zostaną zastąpione lub przekształcą się w wielkie systemy komputerowe, których możliwości będą o rzędy wielkości większe niż jakichkolwiek systemów obecnie dostępnych. Takie systemy zaczynają się już pojawiać, np. systemy bardzo dużych baz danych (rejestry bankowe o zasięgu krajowym, interakcyjne katalogi konsumenckie, systemy informacji o ruchu statków transoceanicznych itp.). Pierwszym krokiem w tym kierunku są także systemy rezerwacji miejsc lotniczych i hotelowych. W końcowej postaci systemy te mogą być używane przez miliony ludzi jednocześnie.

Systemy inteligentne. Inteligentne systemy oprogramowania spowodują takie wzmocnienie potencjału intelektualnego, że w niektórych dziedzinach znacznie wzrośnie produktywność zawodowa. Roboty inteligentne będą przejmować coraz większą część prac przemysłowych, a być może nawet zastąpią nas w obowiązkach domowych. Coraz bardziej złożone narzędzia zmniejszą zapotrzebowanie na programistów, a mogą zwiększyć indywidualne zapotrzebowanie na zrozumienie podstaw techniki komputerowej. Co więcej, dzisiejsze systemy ekspertowe mogą już jutro pełnić rolę wyroczni.

Wpływ na produkt narodowy brutto. Część produktu narodowego brutto (ang. gross national product, GNP) reprezentująca technikę obliczeniową i przetwarzanie informacji — już znacząca — zwiększy się, gdy społeczeństwo stanie się uzależnione od informacji, tak jak od zboża i metalu.

Nawet, jeśli te przewidywania są niedokładne w szczegółach, to nie ulega wątpliwości, że rola komputerów we współczesnym świecie zmieni się znacznie. Z takiego obrazu przyszłości wynikają dalsze konsekwencje.

Zagadnienia konsumenckie. Użycie komputerów przez dużą liczbę osób nie związanych z techniką i wzrost liczby krytycznych zastosowań komputerów spowodują zwiększenie odpowiedzialności wytwórców za wyroby. Z pewnością powstaną problemy analogicznie do znanych dotychczas, związanych z odpowiedzialnością zawodową, handlem, używalnością i niezawodnością, licencjonowaniem, prawami autorskimi i bezpieczeństwem wyrobów, np. należy spodziewać się opracowania atestu analogicznego do wydawanego na wyroby elektryczne. Bez wątpienia, pojawią się też nowe problemy, związane z poufnością i prywatnością, wynikające ze szczególnej natury komputerów.

Produkcja i dystrybucja. Rosnąca rola komputerów i wyrobów komputerowych w handlu detalicznym spowoduje powstanie nowych problemów w wytwarzaniu, sprzedaży i serwisie, normalizacji oraz pobieraniu opłat za współdzielenie zasobów. Inna klasa problemów będzie dotyczyła tego, jak tworzyć oprogramowanie dla masowego odbiorcy. Być może powstaną metody masowej produkcji oprogramowania, np. polegające na dostosowywaniu pakietów lub użyciu oprogramowania do tworzenia oprogramowania, a nie — na tradycyjnym pisaniu kodu.

Poufność i bezpieczeństwo. Oprócz zagadnień związanych z bezpieczeństwem konsumentów, można oczekiwać występowania kwestii dotyczących licencjonowania, odpowiedzialności za produkt i wiarygodności informacji udostępnianej przez publiczne bazy danych. Zwiększy się znaczenie problematyki poufności i prywatności informacji, szczególnie w sytuacjach, gdy będzie chodzić o życie ludzkie.

Skutki ekonomiczne. Należy się spodziewać bardzo rozległych ekonomicznych skutków innowacji technologicznych. Szczególnie istotne dla przemysłu komputerowego będzie współoddziaływanie rozwoju technologicznego i czynników (takich jak produktywność), ograniczających wzrost sektora informacyjnego. Dużą rolę w procesie dojrzewania przemysłu komputerowego będą odgrywały dokładne oceny kosztu oprogramowania i rozważna polityka handlowa. Jedne z najważniejszych przemian ekonomicznych będą dotyczyły personelu, szczególnie w sytuacjach, gdy stanowiska nie wymagające kwalifikacji zostaną wyeliminowane wskutek automatyzacji lub zastąpione przez inne, wymagające wiedzy technicznej na wysokim poziomie.

Skutki psychologiczne. Obecnie ludzie pracują bezpośrednio z komputerami przede wszystkim z wyboru. Wraz z rozpowszechnianiem się komputerów praca z nimi stanie się koniecznością. Spowoduje to bardzo różnorodne skutki dla człowieka, np. konieczność projektowania systemów dla użytkowników bez przygotowania, zmiany rozmieszczenia personelu pod wpływem zmian technologicznych, istotne przemiany w treści i stylu nauczania.

Skutki społeczne. Era komputerów może przyczynić się do powstania nowej, upośledzonej warstwy społecznej — analfabetów komputerowych. Trzon tej nowej warstwy mogą stanowić kobiety i mniejszości narodowe, ze względu na swoje niedostateczne wykształcenie techniczne. Aby nie dopuścić do powstania takiej sytuacji, informatycy muszą uświadamiać sobie skutki społeczne swojej pracy, a społeczeństwo musi być świadome skutków wywoływanych przez tę nową technologię.

Z przedstawionych prognoz wynika, że inżynieria oprogramowania musi być zdolna do zajmowania się problemami związanymi z występowaniem nieprzygotowanych użytkowników, bardzo niejednorodnych systemów i zwiększonych wymagań jakościowych. Poniżej omówiono wynikające stąd zagadnienia oprogramowania.

INŻYNIERIA OPROGRAMOWANIA JAKO NAUKA

Tradycyjne metody wytwarzania oprogramowania są niezorganizowane i polegają na intensywnym wykorzystaniu pracy ludzkiej. Nie są one odpowiednie do spełnienia zwiększonych wymagań nakładanych na systemy komputerowe ani też dostosowane do złożoności systemów finalnych. Inżynieria oprogramowania musi rozwinąć się w kierunku intensywnego wykorzystania technologii opartej na modelach i teoriach mających naukowe podstawy, musi być przygotowana do szybkiego wykorzystania postępów w nauce. Kluczową rolę w tym procesie będzie odgrywać wykształcenie projektantów oprogramowania, ponieważ dobre koncepcje znajdują praktyczne wykorzystanie tylko w rękach tych, którzy je mądrze zastosują.

W ciągu dwóch ostatnich dziesięcioleci takie przemiany w technologii zachodziły pod wieloma względami w programowaniu małoskalowym. Niektóre z wcześniej opracowywanych modeli formalnych zastosowano do analizy algorytmów. Zrozumienie algorytmów rozwiązywania pewnych rodzajów problemów jest obecnie dość dokładne, można analizować działanie określonych algorytmów i w wielu wypadkach określić teoretyczne granice ich funkcjonowania. Podobnie w latach siedemdziesiątych rozwinęła się teoria abstrakcyjnych typów danych. Już w końcu lat sześćdziesiątych informatycy uznali znaczenie dobrej reprezentacji danych i konstruowania odpowiadających im struktur. Udoskonalanie pierwotnych koncepcji aż do powstania teorii abstrakcyjnych typów danych trwało całe dziesięciolecie, ponieważ wymagało równoległego rozwoju specyfikacji formalnych, języków programowania, metod weryfikacji i metodologii programowania. Studenci kierunków informatycznych powinni już teraz rutynowo posługiwać się analizą algorytmów i abstrakcyjnymi typami danych. W chwili obecnej można się spodziewać rozsądnych, choć nie w pełni realistycznych zastosowań tych metod w codziennej praktyce.

Ścisłe teorie mogą też przyczynić się znacznie do ulepszenia metod konstruowania systemów oprogramowania. Przykładowo, kompilator języka programowania jest systemem średniej wielkości, którego struktura jest obecnie dość dobrze zrozumiana. O ile na początku lat sześćdziesiątych skonstruowanie kompilatora było dość znacznym osiągnięciem, to obecnie kompilatory tworzy się standardowo. Dobre podstawy teoretyczne doprowadziły do powstania efektywnych technik konstruowania analizatorów składni (ang. parsers), początkowo ręcznego, a ostatnio — automatycznego. Podobnie rozwój teorii dotyczących semantyki języków programowania prowadzi do automatyzacji kolejnych kroków w konstruowaniu kompilatorów.

Choć programowanie wielkoskalowe ma nieco krótszą historię, zaczynają się już pojawiać metody formalne służące do operowania dużą liczbą informacji w tej dziedzinie. Przykładowo, metody zarządzania konfiguracją i kontroli wersji powstały początkowo na zasadzie ad hoc, dostarczając prostych narzędzi do zorganizowanej i często masowej, ponownej kompilacji modułów. Jednakże, przynajmniej niektóre modele konfigurowania i ponownego wytwarzania systemów wymagają konstruowania narzędzi programowych. Opierając się na podstawach teoretycznych można nie tylko lepiej wykorzystać informację o współzależności modułów do poprawnego, ponownego skonstruowania systemu, ale także zastosować bardziej wydajne strategie ponownego konstruowania systemów dzięki uniknięciu zbędnych kroków, np. ponownej kompilacji modułów, w których zmieniono jedynie komentarze lub które zależą tylko od nie zmienionych części zmodyfikowanych modułów.

Przedstawione przykłady pozwalają na zorientowanie się w skali postępu prowadzącego do oparcia inżynierii oprogramowania na ściślejszych podstawach. Jest jeszcze wiele obszarów, gdzie stosowane modele, teorie i metodologie są dość prymitywne. Jednakże, skuteczność ścisłych teorii, przynajmniej w niektórych dziedzinach, stanowi zachętę do rozwijania i udoskonalania ich w innych dziedzinach.

KOLEJNE WYZWANIA

W ciągu dziesięciolecia, odkąd uznano programowanie wielkoskalowe za znaczący element inżynierii oprogramowania, złożoność systemów oprogramowania wzrosła znowu o rząd wielkości i obecnie obserwuje się kolejną przemianę. Projektanci oprogramowania muszą dziś zajmować się złożonymi systemami, w których oprogramowanie jest tylko jedną z wielu składowych dużej i niejednorodnej całości lub w których od oprogramowania oczekuje się zastąpienia człowieka-programisty i podjęcia aktywnej roli w procesie wytwarzania i kontroli systemu. Tym dwom nowym trybom pracy można nadać, odpowiednio, nazwy: program jako składowa (ang. program-as-component) i program jako zastępca (ang. program-as-deputy). Ich stosunek do programowania mało- i wielkoskalowego, a także do programowania niezorganizowanego lat sześćdziesiątych przedstawiono w tabeli.

Wyodrębnienie tych dwu nowych trybów pracy programu oznacza przemiany w charakterze problemów rozwiązywanych komputerowo, jak również przemiany w charakterze procesu wytwarzania i pielęgnacji oprogramowania:

- rozwiązania niekoniecznie muszą być algorytmiczne, korzystniejsze może być podejście heurystyczne,
- w rozwiązaniu występują elementy podejmowania decyzji, takie jak wybór zróżnicowanych, ale nie bezwzględnych preferencji,
- rozwiązanie problemu zależy od specyficznej o nim wiedzy, a ta powinna być dynamicznie dostarczana,
- problemy są tak złożone, że rozwiązania nie mogą być określone a priori, lecz muszą powstawać w wyniku doświadczenia,
- rozwiązanie wiąże się z integracją niejednorodnego zbioru składowych systemu, zarówno sprzętowych jak i programowych,
- rozwiązanie obejmuje łagodne przyswojenie niepewnych danych i innych niejednoznaczności fizycznego systemu,
- mogą powstawać ograniczenia zewnętrzne, wynikające z fizycznej natury sterowanego systemu, a nie z jego funkcji logicznych.

Rola programu jako składowej powstaje w dużych i niejednorodnych systemach. Są to systemy zawierające programy w wielu językach dla bardzo złożonego sprzętu, który może mieć ograniczenia mechaniczne, wytwarzać fałszywe dane lub narzucać silne uwarunkowania czasowe. Aby uchwycić istotę tej przemiany, należy uwzględnić te same atrybuty co poprzednio.

Problemy charakterystyczne. Centrum uwagi w projektowaniu przesuwa się z algorytmów i sprzężeń na zintegrowanie systemu jako całości.

Główne zagadnienia danych. Pojawia się potrzeba zintegrowanych baz danych, zawierających nie tylko informację symboliczną i numeryczną, lecz także informację o fizycznym stanie systemu, który w rzeczywistości może być fizycznie rozproszony, wysuwając na pierwszy plan kwestie komunikacji.

Główne zagadnienia sterowania. Oprogramowanie musi obecnie zapewniać sterowanie złożonymi systemami, które mogą zawierać dane podlegające ograniczeniom fizycznym lub mechanicznym, jak również zwykłe dane czysto symboliczne.

Zagadnienia specyfikacji. Specyfikacje oprogramowania muszą uwzględniać jego sprzężenia z nieprogramowanymi elementami systemu, jak również z innymi elementami programowymi.

Charakter przestrzeni stanów. Przestrzeń stanów dużego, niejednorodnego systemu może być bardzo duża. Ponadto, jej struktura musi być dynamicznie rekonfigurowana i może zawierać zarówno elementy fizyczne jak i symboliczne.

Zarządzanie. Niejednorodny charakter tych systemów zwiększa wymagania dotyczące zarządzania i koordynacji harmonogramów: projektu, opracowania, konstruowania i integracji.

Narzędzia i metody. Rozważane systemy wymagają sterowania w czasie rzeczywistym i sprzężeń użytkowych dostosowanych do obsługi przez niefachowców. Muszą być zdolne do wykonywania złożonych zadań sterowania, w zasadzie bez interwencji operatora.

Rola programu jako zastępcy powstaje wtedy, gdy duże, twórcze fragmenty procesu produkcji oprogramowania są wykonywane przez inne oprogramowanie. Ten rodzaj przemian zachodzi bardzo powoli od czasu, kiedy pierwsze symboliczne asemblery poczęły przypisywać adresy zmiennym. Wraz z upływem czasu, coraz więcej doświadczeń dotyczących procesu produkcji oprogramowania zmaterializowano w programach, które wykonują coraz bardziej twórcze podzadania procesu produkcji oprogramowania i zarządzania nim. Przekazując te podzadania narzędziom i generatorom programów powoduje się zwiększenie poziomu wielokrotnego używalności oprogramowania, od fragmentów kodu do elementów projektu. Następujące atrybuty charakteryzują te przemiany.

Problemy charakterystyczne. Centrum uwagi kieruje się na włączenie doświadczeń i podejmowania decyzji do narzędzi programowych. Obecne zmiany polegają na próbie uzbrojenia narzędzi automatycznych w możliwości podejmowania decyzji subiektywnych.

Główne zagadnienia danych. Obecnie oprogramowanie wykonuje automatycznie pewne zadania w procesie produkcyjnym oprogramowania, musi więc korzystać nie tylko z reprezentacji danych w dziedzinie zastosowania, lecz także z wiedzy o specyficznym dla tej dziedziny doświadczeniu i stanie procesu produkcji. Co więcej, reprezentację tę należy dobrać tak, aby umożliwić uczenie się, tj. zachowanie informacji o określonych procesach produkcyjnych w celu wykorzystania jej do ogólnego ulepszenia narzędzi.

Główne zagadnienia sterowania. Programy muszą nie tylko zawierać doświadczenie, lecz także uczyć się na podstawie ich poprzedniego użycia.

Zagadnienia specyfikacji. Punkt ciężkości przesuwa się z wielokrotnego użycia kodu na wielokrotne użycie projektu, możliwe dzięki wprowadzeniu automatyzacji.

Pojawianie się nowych zagadnień programistycznych wraz ze wzrostem złożoności systemów

Atrybuty	Programowanie niezorganizowane — od 1960 roku	Programowanie małoskalowe — od 1970 roku	Programowanie wielkoskalowe — od 1980 roku	Program jako składowa — od 1990 roku	Program jako zastępca — od 1990 roku
Problemy charakterystyczne	Małe programy	Algorytmy i programowanie	Sprzężenia, zarządzanie, struktury systemowe	Integracja niejednorodnych składowych	Podejmowanie decyzji
Zagadnienia danych	Reprezentowanie struktury i informacji symbolicznej	Struktury danych i typy danych	Długożyciowe bazy danych — symboliczne i numeryczne	Zintegrowane bazy danych — fizyczne i symboliczne	Reprezentacja wiedzy
Zagadnienia sterowania	Elementarne zrozumienie przepływu sterowania	Programy wykonywane jednarozowo i kończone	Programy wykonywane w sposób ciągły	Sterowanie złożonych systemów fizycznych	Uczenie się programów na podstawie własnego zachowania
Zagadnienia specyfikacji	Mnemoniki, precyzyjne użycie języka naturalnego	Proste specyfikacje wejścia-wyjścia	Systemy o złożonej specyfikacji	Oprogramowanie jako składowa niejednorodnego systemu	Wielokrotne używanie projektów
Przestrzeń stanów	Niezbyt dobre zrozumienie stanu za wyjątkiem sterowania	Mała, prosta przestrzeń stanów	Duża, ustrukturalizowana przestrzeń stanów	Bardzo duża przestrzeń stanów o dynamicznej strukturze	Pojęcie stanu dotyczy zarówno wytwarzania jak i zastosowania
Zarządzanie	Brak	Indywidualne	Zespołowe, pielęgnacja systemu w jego cyklu istnienia	Koordynacja wzajemnych oddziaływań	Wiedza o dziedzinie zastosowań i procesie produkcyjnym
Narzędzia i metody	Asemblery i zrzuty zawartości pamięci	Języki programowania, kompilatory, linkery i programy ładujące	Środowiska programowe, zintegrowane narzędzia, dokumentacja	Narzędzia do sterowania w czasie rzeczywistym, dynamiczne rekonfigurowanie	Generatory programów, systemy ekspertowe, systemy uczące się

Charakter przestrzeni stanów. Pojęcie przestrzeni stanów należy rozszerzyć, aby obejmowało zarówno procesy produkcyjne jak i dziedzinę zastosowań.

Zarządzanie. Należy zdobywać i opanowywać wiedzę jakościową i ilościową o procesie produkcyjnym oprogramowania i o dziedzinie zastosowań.

Narzędzia i metody. Do najwcześniej powstałych silnych narzędzi należą generatory programów. W chwili obecnej dołączają do nich systemy uczące się i systemy reprezentacji wiedzy.

Wymienione zagadnienia obrazują przemiany jedynie w technologii wytwarzania i pielęgnacji oprogramowania. Ze wzrostem skali systemów należy jednak uwzględnić także szereg innych czynników.

Zagadnienia zawodowe. Przez najbliższe 5—10 lat inżynieria oprogramowania doświadczy drastycznego braku specjalistów. Poświęcenie większej uwagi kształceniu i rozwojowi zawodowemu pozwoli na szybsze uzupełnienie braku kadr.

Zagadnienia prawne. Oprogramowanie nie jest podobne ani do wyrobów sprzętowych, ani do książek. Wskutek tego ani prawa patentowe, ani prawa autorskie nie mają zastosowania do produktów i narzędzi programowych. Prawo ochrony własności intelektualnej w zastosowaniu do oprogramowania, oprócz samej ochrony oprogramowania, musi uwzględniać takie zagadnienia jak odpowiedzialność handlowa, utrudnienia w upowszechnianiu nowej technologii i prawa do danych technicznych.

Zagadnienia ekonomiczne. Na koszt produkcji oprogramowania ma wpływ wiele różnych czynników, a coraz większą część zasobów przedsiębiorstwa pochłania samo oprogramowanie. Projektanci oprogramowania często nieumiejętnie szacują koszty składowych innych niż te, które są bezpośrednio związane z wytwarzaniem oprogramowania. Z drugiej strony mechanizm zwrotu kosztów wskutek sprzedaży oprogramowania jest niedoskonały, szczególnie w wypadkach, gdy oprogramowanie podlega modyfikacji w celu wielokrotnego użycia. Ponadto znaczny wpływ na decyzje przedsiębiorstw dotyczące polityki innowacyjnej mają zasady podatkowe.

Zagadnienia zarządzania. Problematyka zarządzania stała się istotna w technologii oprogramowania, odkąd rozpoznano znaczenie programowania wielkoskalowego. Wraz ze wzrostem skali systemów metody zarządzania powinny uwzględniać także udoskonalanie technik oceny kosztów, wgląd w proces produkcji oprogramowania w celu zwiększenia efektywności sterowania, opracowanie odpowiednich miar jakości oddziaływań organizacyjnych i miar zmniejszania ryzyka w celu zachęty do wprowadzania bardziej produktywnej technologii programowania.

Mimo że żadne z wymienionych zagadnień dotychczas nie było uwzględniane w inżynierii oprogramowania, współcześnie ich rola niepomiernie wzrasta.

* * *

Zagadnienia i ich rozwiązania omówione powyżej obejmują nieco większy obszar niż rozumiana tradycyjnie inżynieria oprogramowania. Dotyczy to zwłaszcza problemów ostatecznego użytkownika, takich jak jakość produktów konsumpcyjnych czy rola oprogramowania jako integralnej i wyspecjalizowanej składowej większego systemu. Materiał ten zawiera także rozważania dotyczące innych dziedzin informatyki, jak sztuczna inteligencja lub architektura komputerów. Aby inżynieria oprogramowania mogła stawić czoła problemom, które wystąpią w następnym dziesięcioleciu, powinna uwzględnić zwiększenie ich skali i zakresu:

1) wielkość i złożoność systemów będzie ciągle wzrastać, należy zatem poszukiwać sposobów włączenia oprogramowania jako partnera do procesu jego własnej produkcji;

2) technika komputerowa będzie odgrywać coraz większą rolę we wszystkich dziedzinach, należy więc stopniowo patrzeć na oprogramowanie jak na jedną ze składowych niejednorodnego, lecz zintegrowanego systemu.

Inżynieria oprogramowania powinna być dziedziną na tyle szeroką, aby obejmować zagadnienia projektowania i integracji oprogramowania oraz procesy jego produkcji i pielęgnacji, włączając zarówno elementy techniczne jak i nietechniczne. Powinna uwzględniać przesunięcie centrum uwagi na nowe zagadnienia, związane z rolą programów jako składowych i programów jako zastępców, pojawiające się wraz z rozwojem zastosowań i powstawaniem nowych problemów.

*

Przedstawione rozumienie inżynierii oprogramowania jest wynikiem wielu dyskusji autorki z innymi informatykami, zwłaszcza z kolegami z Uniwersytetu Carnegie-Mellon. Szczególny wpływ na wyrażone tu poglądy mieli: Bill Wulf, Allen Newell, Nico Habermann i Jim Horming.

Oprócz literatury podanej w bezpośrednich odsyłaczach w pierwszej części, korzystano z kilku innych źródeł. Materiał z punktu „Istota problemu oprogramowania" przygotowano do publikacji [1], a treść punktów „Rozszerzenie zakresu techniki obliczeniowej" i „Inżynieria oprogramowania jako nauka" pochodzi z publikacji [3] i [4]. Poglądy autorki na istotę problemu oprogramowania ukształtowały się w znacznym stopniu w ciągu trzydniowego seminarium, którego wyniki przedstawiono w raporcie [2].

LITERATURA

[1] Barbacci M., Habermann A. N., Shaw M.: The Software Engineering Institute — Bridging Practice and Potential. IEEE Software, Vol. 2, No. 6, pp. 4—21, November 1985
[2] Musa J. D.: Software Engineering — The Future of a Profession. IEEE Software, Vol. 2, No. 1, pp. 55—62, January 1985
[3] Shaw M. (ed.): The Carnegie-Mellon Curriculum for Undergraduate Computer Science. Springer-Verlag, New York, 1985
[4] Shaw M.: Education for the Future of Software Engineering. Proc. of Software Engineering Institute Software Engineering Education Workshop, Springer-Verlag, New York, 1986.

Tłum. i oprac.
JANUSZ ZALEWSKI

Zastosowanie Prologu w bazach danych (1)

dokończenie ze str. 4

[10] Ioannidis Y. E., Shinkle L. L., Wong E.: Enhancing INGRES with Deductive Power [12]
[11] Jarke M., Linnemann V., Schmidt J. W.: Data Constructors — on the Integration of Rules and Relations. Proc. of 11th Conf. of VLDB, Stockholm, 1985
[12] Kerschberg L. (ed.): Proc. of 1st Intern. Workshop on Expert Database Systems, Kiawah Island, 1984
[13] King J. J.: A System for Semantic Query Optimization in Relational Databases, Proc. of 7th Conf. of VLDB, Cannes, 1981
[14] Kowalski R. A.: Predicate Logic as Programming Language. Proc. of IFIP Congress, North-Holland, Amsterdam, 1974
[15] Kluźniak F., Szpakowicz S.: Prolog, WNT, Warszawa, 1983
[16] Kluźniak F., Szpakowicz S.: Prolog for Programmers. Academic Press, New York, 1985
[17] Lloyd J. W.: Foundation of Logic Programming. Springer-Verlag, Berlin, 1984
[18] Lloyd J. W., Topor R. W.: A Basis for Deductive Database Systems. Journal of Logic Programming, Vol. 2, No. 2, 1985
[19] Robinson J. A.: A Machine Oriented Logic Based on The Resolution Principle. Journal of ACM, Vol. 1, No. 14, 1965
[20] Stonebraker M.: Inference in Database Systems Using Lazy Triggers [2]
[21] Ullman J. D.: Principles of Database Systems. Computer Science Press, 1982
[22] Ullman J. D.: Implementation of Logical Query Language for Databases. ACM Trans. on Database System, Vol. 10, No. 3, 1985
[23] Zaniolo C.: The Representation and Deductice Retrieval of Complex Objects. Proc. of 11th Conf. of VLDB, Stockholm, 1985.

Struktura systemu operacyjnego PC-DOS (2)

dokończenie ze str. 18

LITERATURA

[1] Grabowicz R., Zalewski J.: System operacyjny PC-DOS, cz. 4, Informatyka, nr 1, str. 21—23, 1987
[2] Markowski M. A.: Logiczna organizacja dysku w systemie operacyjnym PC-DOS. Informatyka, nr 7—8, str. 38—39, 1986
[3] Zalewski J.: Porównanie systemów BIOS dla rodziny Komputerów IBM PC. Informatyka, nr 4, str. 18—21, 1987.

RYSZARD TADEUSIEWICZ
Instytut Automatyki
Akademia Górniczo-Hutnicza
Kraków

Głosowe wprowadzanie informacji do komputera (2)

W poprzedniej części artykułu przedstawiono ogólne założenia dotyczące systemu automatycznego rozpoznawania mowy, który może być wykorzystywany jako wygodne i bardzo naturalne dla człowieka urządzenie wejściowe w niektórych typach systemów komputerowych. Wskazano na wieloetapowy charakter procesu rozpoznawania mowy, a także zasygnalizowano główne kierunki prac, prowadzonych na świecie i w Polsce w dziedzinie rozpoznawania sygnału mowy.

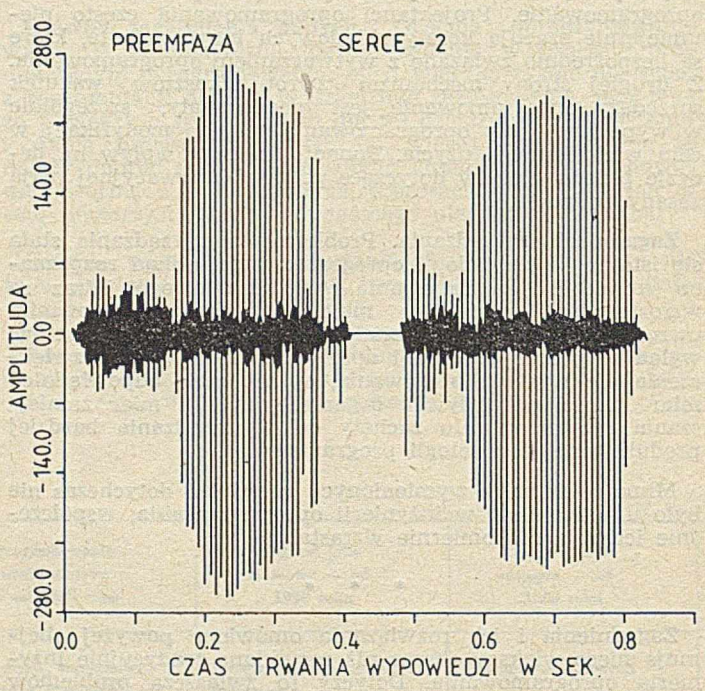
Poniżej szczegółowo przedstawiono zestaw operacji i przekształceń, jakim należy poddać sygnał mowy w trakcie jego wprowadzania do komputera, przetwarzania i rozpoznawania. Artykuł oparto na badaniach autora i współpracowników z Instytutu Automatyki AGH. Podana niżej koncepcja systemu rozpoznawania mowy polskiej nie jest kompletna w tym sensie, że nie udało się dotychczas zrealizować i połączyć ze sobą wszystkich wymienionych modułów systemu i nie ma zatem fizycznie zrealizowanego systemu zdolnego do pełnego rozpoznawania mowy polskiej. Systemu takiego nie zdołano zrealizować dla żadnego z istniejących języków, chociaż prace dla języków angielskiego i japońskiego są bez wątpienia dalej posunięte niż badania nad rozpoznawaniem języka polskiego.

Wprowadzanie sygnału mowy do komputera

Wprowadzanie różnych sygnałów do komputera, który je analizuje i przetwarza, jest czynnością dobrze znaną i nie są tajemnicą konieczne do wykonania operacje filtracji, próbkowania (z ewentualnym uśrednianiem), kompresji dynamiki sygnału i wreszcie konwersji analogowo-cyfrowej z użyciem przetwornika o wynikającej z wymaganej dokładności liczbie bitów sygnału wyjściowego. Wydaje się więc, że nie ma sensu opisywanie i oddzielne rozważanie wprowadzania do komputera sygnału mowy, który jest wszak dość typowym i z pozoru wcale nie trudnym w analizie sygnałem akustycznym o właściwościach dobrze znanych z telefonii, radiofonii oraz innych technik rejestracji dźwięków.

Rzeczywistość jest jednak bardziej skomplikowana. Sygnał mowy ma — wbrew pozorom — dość szerokie pasmo częstotliwości. Wprawdzie w komunikacji między ludźmi można rozpatrywać ten sygnał w ograniczonym pasmie nie ograniczając jego zrozumiałości (stosuje się pasmo telefoniczne 300—3500 Hz przy przesyłaniu sygnału drogą kablową lub radiową, uzyskując zadowalającą jakość transmisji), jednak do automatycznego rozpoznawania mowy celowe okazuje się wykorzystywanie sygnału o znacznie szerszym pasmie (do kilkunastu kiloherców) — z oczywistą konsekwencją w postaci liczby próbek, które trzeba w jednostce czasu wprowadzić do komputera w celu prawidłowego odwzorowania sygnału. Podobnie zakres amplitud sygnału jest bardzo szeroki. W telefonii cyfrowej wykorzystuje się przetworniki 8-bitowe (na przykład w tzw. kodekach), ale w analizie i rozpoznawaniu mowy okazuje się to liczbą niewystarczającą, zwłaszcza do skutecznego przenoszenia zarówno wysokoenergetycznych składników sygnału mowy (głównie samogłosek), jak i składowych niskoenergetycznych (na przykład spółgłoski F lub H). Duże wymagania w dziedzinie częstotliwości i amplitud rozważanego sygnału pociągają za sobą duże wymagania dotyczące pamięci komputera rejestrującego wprowadzany do niego bezpośrednio sygnał mowy (rys. 1).

Przykładowo, pamięć rzędu 64 KB wystarcza (przy zachowaniu dużej wierności odwzorowania) do zarejestrowania około jednej sekundy trwania sygnału. Jest to nie do przyjęcia z trzech powodów. Po pierwsze — pamięć komputera jest nadal jednym z najdroższych jej elementów. Przeznaczanie tak dużych obszarów pamięci na rejestrację sygnału (nie licząc obszaru koniecznego do zarejestrowania programu przetwarzania i rozpoznawania sygnału, a także innych programów) jest nieuzasadnione. Po drugie — przy tak dużej objętości informacyjnej zarejestrowanych danych procedury przetwarzania i rozpoznawania sygnału będą wymagały bardzo długiego czasu przetwarzania — bez względu na to, jak sprytne algorytmy i programy uda się ułożyć. Osiągnięcie w tych warunkach pracy w czasie rzeczywistym wydaje się mało realne — nawet przy posługiwaniu się bardzo dużymi i szybkimi komputerami. Kto zaś ma ochotę marnować możliwości dużego komputera na realizację operacji — w całym tego słowa znaczeniu — pomocniczych? Rozpoznawanie mowy powinien realizować (docelowo, po dostatecznym opracowaniu metod) specjalizowany procesor, wchodzący w skład peryferyjnego wyposażenia komputera.

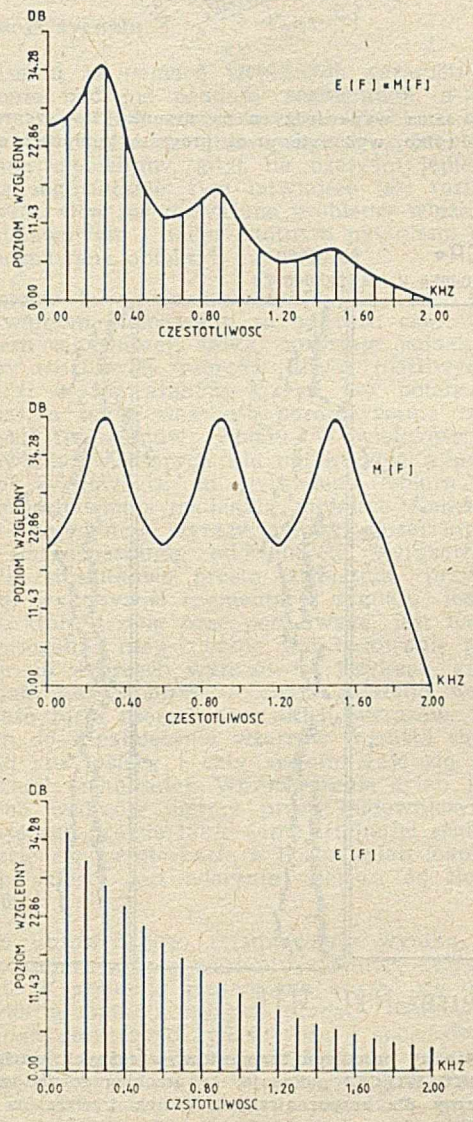


Rys. 1. Przebieg czasowy sygnału mowy (wypowiedź SERCE); ta postać prezentacji sygnału jest mało przydatna z punktu widzenia rozpoznawania mowy, a zajmuje dużo miejsca w pamięci komputera

Trzeci powód wiąże się z samymi metodami rozpoznawania. Można wykazać, że w sygnale mowy, nawet bardzo dokładnie i wiernie odwzorowanym w pamięci komputera, cechy przydatne do rozpoznawania i cechy utrudniające ten proces są tak dokładnie ze sobą splecione, iż rozdzielenie ich napotyka poważne trudności. W szczególności przy czasowej reprezentacji sygnału mowy przebieg sygnału bywa silniej determinowany przez właściwości osobnicze, niż przez treść wypowiedzi. Można się o tym przekonać porównując oscylogramy sygnału mowy, rejestrowane dla różnych wypowiedzi i różnych osób. Często bywa tak, że zapisy odpowiadające różnym słowom wypowiadanym przez tę samą osobę wykazują większe wzajemne podobieństwo, niż wypowiedzi tego samego słowa pochodzące od różnych osób. W dodatku przebieg czasowy sygnału mowy jest tyl-

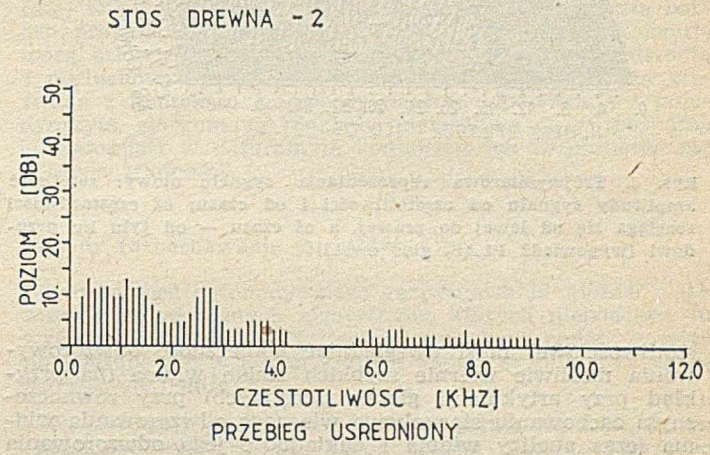
ko częściowo zdeterminowany przez treść wypowiadanych wiadomości i przez cechy osobnicze głosu osoby mówiącej. Znaczny udział w kształtowaniu czasowego przebiegu sygnału konkretnej wypowiedzi mają czynniki losowe, związane z procesem generowania dźwięków w narządach mowy.

W dużym uproszczeniu można powiedzieć, że proces artykulacji mowy składa się z dwu procesów: tworzenia dźwięku i jego modulacji (rys. 2). Modulacja, kształtowana świadomie przez ruchy narządów artykulacyjnych, pozostaje pod kontrolą osoby mówiącej i dlatego w niej głównie manifestują się semantyczne treści wypowiadanych słów. Natomiast tworzenie dźwięku, będącego później obiektem modulacji, zachodzi w krtani (drgania strun głosowych) i w przewężeniach narządów mowy (szum przeciskającego się powietrza), przy czym szczegóły tego procesu nie podlegają tak precyzyjnej kontroli, a właściwości tworzonego sygnału (na przykład jego faza) nie są zupełnie zdeterminowane i mają cechy typowego procesu stochastycznego. W rezultacie przy rozpoznawaniu mowy składniki sygnału, pochodzące od procesu generowania dźwięku, utrudniają wydobycie i właściwe interpretowanie tych składników, które powstają w wyniku modulacji, a forma sygnału w postaci przetworzonego cyfrowo i wprowadzonego do komputera przebiegu czasowego jest tu szczególnie niekorzystna.



Rys. 2. Widmo sygnału mowy (u góry) powstaje przez nałożenie charakterystyki amplitudowo-częstotliwościowej kanału głosowego (w środku) na widmo tonu krtaniowego (u dołu); świadomie kształtowanym składnikiem jest przy tym jedynie charakterystyka kanału głosowego, zależna od ruchów artykulacyjnych narządów mowy

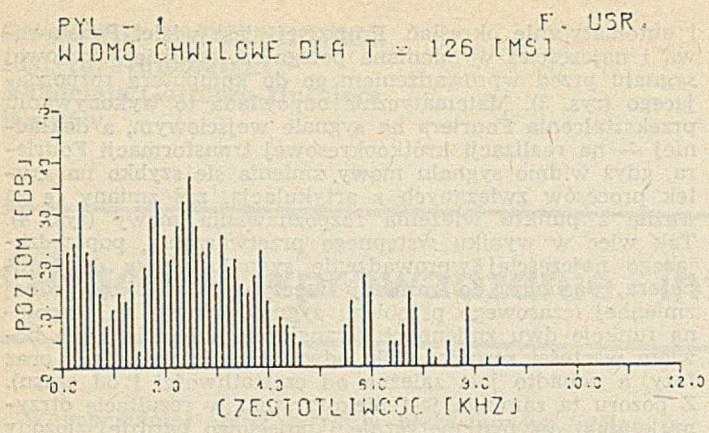
Ustaliwszy celowość poszukiwania innych właściwości sygnału mowy niż zwyczajne wierne odwzorowywanie w komputerze czasowego przebiegu sygnału, należy odpowiedzieć na pytanie — jakie to mają być właściwości, a także jak je przy wprowadzaniu sygnału do komputera wydobywać i automatycznie określać. Propozycji jest wiele. Podstawowa i najczęściej wymieniana polega na analizie widmowej sygnału przed wprowadzeniem go do komputera rozpoznającego (rys. 3). Matematycznie odpowiada to wykonywaniu przekształcenia Fouriera na sygnale wejściowym, a dokładniej — na realizacji krótkookresowej transformacji Fouriera, gdyż widmo sygnału mowy zmienia się szybko na skutek procesów związanych z artykulacją, zaś zmiany te są ważne z punktu widzenia rozpoznawania mowy (rys. 4). Tak więc w wyniku wstępnego przetwarzania, poprzedzającego najczęściej wprowadzenie sygnału mowy do komputera, dochodzi do zamiany rzeczywistej funkcji jednej zmiennej (czasowego przebiegu sygnału mowy) na zespoloną funkcję dwu zmiennych (transformata Fouriera ma bowiem wartości zespolone dla odwzorowania amplitudy oraz fazy, a ponadto jest zależna od częstotliwości i od czasu). Z pozoru ta zamiana jest niekorzystna: w rezultacie otrzymuje się twór matematycznie i pojęciowo bardziej złożony niż sygnał pierwotny (rys. 5). Jest to jednak wrażenie mylące. Można wykazać, że zasadnicze (ważne z punktu widzenia rozpoznawania) cechy sygnału są zawarte w jego widmie amplitudowo-częstotliwościowym; widmo to jest symetryczne (sygnał wejściowy jest rzeczywisty) i może być reprezentowane w komputerze z wykorzystaniem znacznie mniejszej pamięci niż sygnał źródłowy. Warto w tym miejscu wspomnieć, że przekształcenia, jakim podlega sygnał dźwiękowy w systemie słuchowym człowieka, dają się też opisać jako transformacja Fouriera z wydobyciem składowej amplitudowo-częstotliwościowej. Głównie ta właśnie informacja jest przekazywana do wyższych pięter systemu słuchowego i głównie ona decyduje o sposobie słyszenia dźwięków.



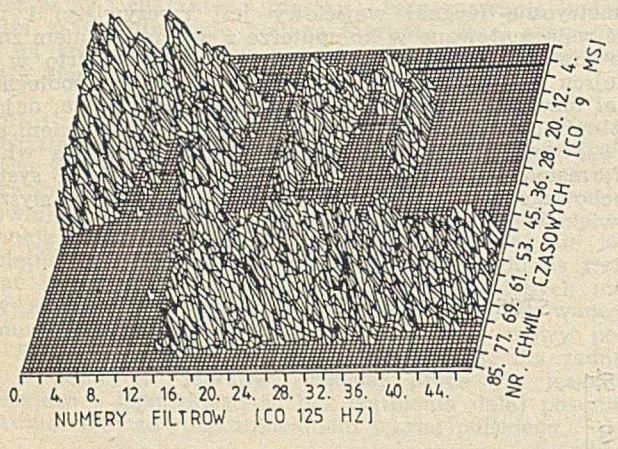
Rys. 3. Widmo sygnału uśrednione na długości całej wypowiedzi zawiera zbyt mało charakterystycznych szczegółów, by mogło być użyteczne do automatycznego rozpoznawania (na rysunku uśredniony przebieg widma wypowiedzi STOS DREWNA)

Jest oczywiście sprawą otwartą, jak należy realizować wymaganą transformację w konkretnym rozwiązaniu sprzętowym. Można tu decydować się na rozwiązania czysto cyfrowe (filtry cyfrowe o odpowiednich charakterystykach lub procesory szybkiej transformaty Fouriera), można także decydować się na rozwiązanie hybrydowe, wprowadzając do systemu elementy analogowe w postaci zestawu odpowiednich filtrów pasmowych, demodulatorów, układów przetwarzających itd. W krajach zachodnich, gdzie elementy cyfrowe (także specjalizowane procesory sygnałowe) są łatwo dostępne, dominują rozwiązania cyfrowe, w opracowaniach krajowych natomiast zwykle wybierano rozwiązania hybrydowe.

Decydując się na wprowadzanie sygnału mowy w postaci czasowo-częstotliwościowego amplitudowego widma sygnału trzeba ustalić jego parametry: pasmo rozważanych częstotliwości, sposób podziału tego pasma na poszczególne kanały i szerokość pasma odpowiadająca jednemu kanałowi, częstotliwość próbkowania sygnału, dokładność odwzorowania amplitudy itd. Każdy z tych parametrów musi być wybierany kompromisowo, gdyż w każdym bez wyjątku wypadku istnieją przynajmniej dwa sprzeczne uwarunkowania wpływające na wybór tej, a nie innej wartości. Przykładowo, wybór częstotliwości próbkowania sygnału (rozumianego już jako dyskretne widmo amplitudowo-czę-



Rys. 4. Widmo charakterystyczne dla głoski Y

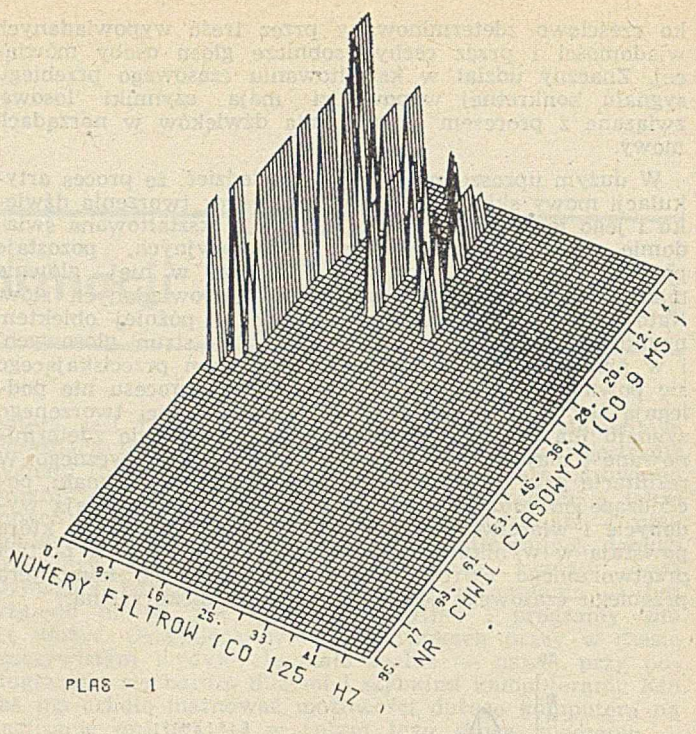


Rys. 5. Trójwymiarowa reprezentacja sygnału mowy: zależność amplitudy sygnału od częstotliwości i od czasu; oś częstotliwości rozciąga się od lewej do prawej, a oś czasu — od tyłu ku przodowi (wypowiedź PLĄS, głos męski)

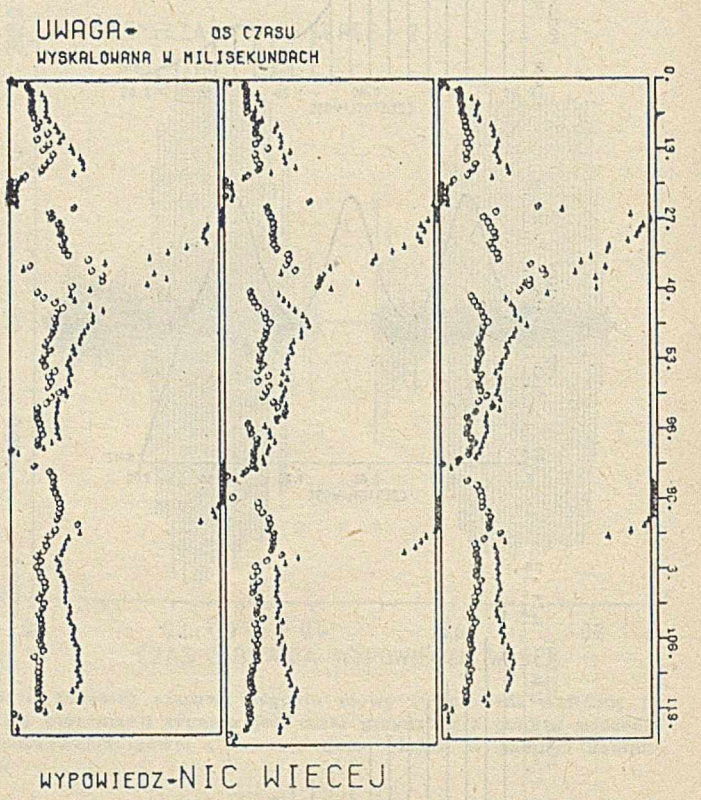
stotliwościowe) musi uwzględniać konieczność odwzorowywania możliwie wiernie szybkich zmian widma (na przykład przy artykulacji głosek plozyjnych) przy równoczesnym zachowaniu stosunkowo wiernego odwzorowania widma (czas analizy widma i dokładność jego odwzorowania pozostają w stosunku odwrotnie proporcjonalnym) oraz przy stałej trosce o wielkość zajmowanej pamięci komputera. Zamiast więc szczegółowo dyskutować wszystkie „za" i „przeciw" dla każdego parametru z osobna, lepiej podać konkretne właściwości systemu zbudowanego przez autora i wykorzystywanego z powodzeniem przy badaniach nad sygnałem mowy. System ten analizuje pasmo częstotliwości 125—12 000 Hz podzielone w sposób równomierny na 96 kanałów (szerokość kanału — 125 Hz), próbkowane z wykorzystaniem kroku czasowego wynoszącego 10 ms i przetwarzane amplitudowo z dokładnością do 0,5 dB. Wszystkie wyniki prezentowane na rysunkach ilustrujących artykuł pochodzą z tego systemu.

Wydobywanie cech sygnału

Po wprowadzeniu sygnału do komputera zachodzi potrzeba przetworzenia go do postaci użytecznej z punktu widzenia wykorzystywanych algorytmów rozpoznawania. Oznacza to zazwyczaj, że trzeba wydobyć z sygnału te jego cechy, które w sposób stabilny i niezależny od czynników ubocznych (na przykład cech indywidualnych głosu osoby mówiącej) determinują treść wypowiadanych słów. Cechy te ponadto muszą być zdefiniowane w taki sposób, by istotnie było możliwe ich automatyczne wyznaczanie przez komputer. To ostatnie zastrzeżenie jest o tyle istotne, że w literaturze dotyczącej automatycznego rozpoznawania mowy są niekiedy dyskutowane zestawy tak zwanych cech dystynktywnych, które w pełni charakteryzują rozpoznawane elementy (na przykład fonemy), lecz nie daje się ich efektywnie wyznaczyć na drodze analizy zarejestrowanych próbek sygnału.



Rys. 6. Ta sama wypowiedź co na rysunku 5 — reprezentowana przez same tylko, wydobyte przez program komputerowy formanty



Rys. 7. „Mapka" przebiegu formantów w różnych wariantach wypowiedzi NIC WIĘCEJ pokazuje, że przebieg formatów jest charakterystyczny dla rozpoznawanych głosek i wyrazów

Wielu autorów proponuje wykorzystanie różnych cech sygnału i różnych metod ich wyznaczania, jednak najczęściej wskazuje się na celowość posługiwania się tak zwanymi formantami. W przybliżeniu zadowalającym dla celów tego opracowania można je określić jako częstotliwość, wokół których grupuje się lokalnie maksymalna część energii widma. Odpowiada to pojęciu biegunów transmitancji toru głosowego i pozwala dość dobrze odwzorowywać położenie narządów artykulacyjnych i zmiany tego położenia

podczas wypowiadania poszczególnych głosek (rys. 6). Wydzielanie formantów jest możliwe, chociaż wymaga dość przemyślanych algorytmów w celu uniknięcia efektu wykrywania każdego (być może przypadkowego) maksimum obwiedni widma jako formantu. Oprócz formantów możliwe i celowe jest analizowanie innych cech sygnału, zwłaszcza w odniesieniu do głosek szumowych. W pracach autora i współpracowników sugerowano wielokrotnie użycie w tym celu momentów widmowych, które dobrze opisują globalny kształt widma, na przykład jego „środek ciężkości" oraz „szerokość".

Można oczywiście rozważać także użycie innych parametrów, na przykład współczynników predykcji liniowej lub parametrów charakteryzujących cepstrum sygnału. W wyniku określenia wybranych parametrów i wyznaczenia ich wartości w konkretnym wypadku rozważanego fragmentu sygnału mowy następuje przypisanie każdej próbce sygnału określonego punktu w przestrzeni, której osie są wyznaczone przez wyodrębnione cechy. Należy oczekiwać, że przy prawidłowym wyborze cech, punkty odpowiadające tym samym (pod względem treści) fragmentom wypowiedzi będą sąsiadowały ze sobą w przestrzeni cech, a punkty odpowiadające różnym głoskom będą od siebie oddalone. W wypadku przestrzeni cech, zawierającej trzy pierwsze formanty i dwa momenty widmowe, istotnie obserwuje się takie zjawisko (rys. 7).

Segmentacja sygnału

Po opisaniu za pomocą wybranych parametrów, sygnał mowy może być już poddany segmentacji, a powstające segmenty mogą być rozpoznawane. Problem segmentacji jest przy tym nierozerwalnie związany z problemem rozpoznawania elementów, gdyż na ogół im lepiej przeprowadzi się segmentację, tym łatwiejsze jest rozpoznawanie i na odwrót. Oba zaś rozważane problemy wiąże się z bardziej podstawowym i nader istotnym pytaniem: czym mają być rozpoznawane obiekty?

Zagadnienie to warto rozważyć bliżej. Jeśli przyjąć, że rozpoznawanymi elementami są od razu całe wypowiedzi, to problem segmentacji znika, pozostaje jednak zagadnienie liczby możliwych wzorców (liczba możliwych różnych wypowiedzi w rozważanym języku jest potencjalnie nieskończona), a także złożoność porównywania ze wzorcem tak dużych fragmentów sygnału. W większości systemów rozpoznawania, w których nie ograniczono a priori liczby możliwych wypowiedzi do wybranego zbioru umownych haseł, rozpoznawaniu podlegają mniejsze elementy wypowiedzi — pojedyncze wyrazy, sylaby, głoski lub ich fragmenty. Rozpoznawanie oddzielnych wyrazów pozwala wprawdzie stosunkowo prosto (zwłaszcza przy starannej wymowie) wykonywać segmentację sygnału, jednak liczba wzorców, które trzeba brać pod uwagę, jest nadal bardzo duża. Szczególnie niewdzięczne jest bazowanie przy rozpoznawaniu na wzorcach wyrazów w językach, które — podobnie jak język polski — mają składnie wymuszającą odmienność form niemal wszystkich wyrazów. Liczba koniecznych do zapamiętania wzorców rozrasta się przy tym do ponad stu tysięcy i cały system staje się niemożliwy do realizacji technicznej. Wprowadzenie sylab jako rozpoznawanych wzorców piętrzy przed budowniczym systemu rozpoznającego prawie takie same trudności segmentacyjne, jak oparcie się na głoskach, a liczba sylab (około 2500 dla języka polskiego) jest znacznie większa od liczby głosek (około 40).

Z tego uproszczonego rozumowania wynika, że celowe jest rozważenie takiego wariantu systemu, który rozpoznaje głoski. W systemie tym można oprzeć się na licznych, dostępnych w opublikowanych pracach naukowych, wynikach badań dotyczących właściwości i sposobów rozpoznawania głosek (separowanych „ręcznie" z różnych kontekstów lub wypowiadanych w izolacji). Niestety, względna łatwość rozpoznawania głosek nie przesłania bardzo poważnej trudności, jaką jest automatyczna segmentacja ciągłego sygnału mowy na poszczególne głoski. W gruncie rzeczy nie zaproponowano dotychczas naprawdę skutecznego algorytmu takiej segmentacji, a znane próby opracowania programu automatycznej segmentacji zakończyły się albo niepowodzeniem (program wykrywał granice segmentów w niewłaściwych miejscach), albo prowadziły do tak długich czasów obliczeń, że ich przydatność była w najwyższym stopniu wątpliwa.

W tej sytuacji należałoby znaleźć takie segmenty, na które rozpada się on w sposób naturalny, i które jednocześnie umożliwiałyby konstruowanie poprawnych rozpoznań głosek, sylab, wyrazów, wypowiedzi. Można tu wykorzystać fakt, że wprowadzanie sygnału do komputera jest związane z segmentowaniem go na odcinki czasowe (w konkretnym systemie, do którego nawiązuje się w tym artykule, co 9 ms są wprowadzane do komputera kolejne, 96-elementowe widma chwilowe sygnału). Opracowując odpowiednio te kwanty czasowe można znaleźć elementy, stanowiące „wspólne wzorce" dla pewnych klas podobnych do siebie widm chwilowych. Na podstawie żmudnych badań, podczas których wykonano grupowanie z wykorzystaniem algorytmów analizy skupień kilkudziesięciu tysięcy widm chwilowych, udało się wskazać takie wzorce, którym następnie można było przypisać odpowiednie zaszeregowanie w kontekście rozpoznawania głosek.

Udało się wykazać, że istnieje kilkanaście wzorców widm, które są charakterystyczne dla konkretnych, określonych głosek (na ogół jedna głoska, zależnie od kontekstu, jest reprezentowana przez kilka oddzielnych wzorców). Kilkadziesiąt innych wzorców widm charakteryzuje kilka różnych głosek, przy czym na ogół kontekst pozwala takie „wariantowe" wzorce jednoznacznie zaklasyfikować. Kolejnych kilkadziesiąt wzorców można z kolei zweryfikować jako charakterystyczne stany przejściowe między określonymi głoskami — co ułatwia identyfikację. Oczywiście nie wszystkie widma chwilowe dadzą się w tej klasyfikacji jednoznacznie pomieścić. Niektóre mają tak dalece indywidualny charakter, że odbiegają od wszystkich wzorców i muszą być klasyfikowane jako nieznane. Nie stanowi to jednak istotnej przeszkody przy rozpoznawaniu wypowiedzi, gdyż typowa, kilkusłowowa komenda jest rozbita na ponad tysiąc widm chwilowych i nawet w wypadku, gdy kilkadziesiąt spośród nich zostanie rozpoznanych błędnie, możliwe jest odtworzenie poprawnej formy rozpoznania całej wypowiedzi przez analizę kontekstu. Jest zresztą bardzo prawdopodobne, że na podobnej zasadzie funkcjonuje mózg człowieka rozpoznającego mowę, gdyż doświadczenia z odsłuchowym rozpoznawaniem mowy wykazały, że wyrwane z kontekstu fragmenty nagrań odpowiadające pojedynczym głoskom są niekiedy niemożliwe do rozpoznania — natomiast w naturalnym kontekście ich rozpoznanie nie sprawia trudności.

Metody rozpoznawania sygnałów

Sam proces automatycznego rozpoznawania można realizować według różnych algorytmów, których niepodobna tu wyczerpująco omówić. Najczęściej spotykane podejścia można pogrupować w trzech klasach metod, skrótowo zasygnalizowanych poniżej.

W pierwszej grupie metod wykorzystuje się fakt, że po określeniu cech sygnału, każda próbka mowy — zarówno podlegająca rozpoznaniu, jak i stanowiąca wzorzec rozpoznawanego elementu — jest reprezentowana przez punkt w przestrzeni cech lub przez pewien zbiór punktów. Interpretacja jest prosta i naturalna: określiwszy wszystkie cechy rozważanego fragmentu sygnału mowy można cechy te odłożyć na kolejnych osiach przestrzeni o takiej liczbie wymiarów, jak liczba rozpatrywanych cech — w ten sposób zostaje wyznaczony punkt. Bardziej złożonym próbkom sygnału (na przykład całym wyrazom) odpowiadają trajektorie tworzone z kolejno wyznaczonych punktów lub obszary, w których znajdowały miejsce różne realizacje określonego obiektu. Po zarejestrowaniu nowego obiektu, który trzeba rozpoznać, można określić odległość tego obiektu od wszystkich obiektów zapamiętanych jako wzorce i zakwalifikować go do najbliższego wzorca. W ten sposób można identyfikować poszczególne głoski lub ich fragmenty, sylaby lub całe wyrazy.

W wypadku identyfikowania wyrazów lub dłuższych elementów sygnału mowy pojawia się problem niezgodności skal czasowych w wypowiedzi rozpoznawanej i we wzorcu. Tempo mówienia jest zwykle nierównomierne, co powoduje, że nawet przy jednakowej całkowitej długości czasu trwania rozpoznawanej wypowiedzi oraz weryfikowanego wzorca, pojawiają się rozbieżności w zakresie czasu trwania poszczególnych głosek. Pokonanie tego problemu jest możliwe za pomocą metod nieliniowej normalizacji czasowej, opartych koncepcyjnie na metodach programowania dynamicznego, opracowanych pierwotnie dla zadań optymalizacji.

Inna koncepcja rozpoznawania bazuje na pojęciu tzw. funkcji przynależności. Mając n cech, stanowiących pod-

stawę rozpoznawania, można zbudować tyle funkcji n zmiennych, ile jest klas podlegających rozpoznawaniu. Dla każdego obiektu (w tym także dla podlegającego rozpoznawaniu w danej chwili elementu sygnału mowy) można obliczyć wartości wszystkich tych funkcji, gdyż wraz z obiektem są dostępne wartości wszystkich jego cech, a zatem można — przez proste podstawienie — określić wartości funkcji przynależności. Funkcji przynależności jest tyle, ile podlegających rozpoznawaniu klas, można więc ustalić ich przyporządkowanie do poszczególnych klas (na przykład do poszczególnych głosek) i przyjąć zasadę, że nieznany obiekt należy zaliczyć do tej klasy, której funkcja przynależności ma największą wartość po podstawieniu argumentów wynikających z wartości cech nieznanego obiektu. Oczywiście na to, by podana reguła mogła sensownie funkcjonować, są potrzebne metody stosownego budowania funkcji przynależności. Odpowiednie metody są znane — chociaż oczywiście nie można ich tu streścić ze względu na złożoność i liczne a ważne szczegóły.

Trzecia metoda rozpoznawania bazuje na rachunku prawdopodobieństwa. Punktem wyjścia są w niej oszacowania prawdopodobieństw poszczególnych klas (w wypadku głosek mowy dobrze znane), a także gęstości prawdopodobieństwa cech w poszczególnych klasach. Na tej podstawie można za pomocą dość prostych obliczeń określić dla każdego nieznanego obiektu (podlegającego rozpoznaniu wypowiedzi) prawdopodobieństwo jej przynależności do każdej z klas. Dysponując tymi prawdopodobieństwami można potem podejmować dość precyzyjne decyzje: wybrać jedną klasę, jeśli odpowiadające jej prawdopodobieństwo jest dostatecznie wysokie i dominuje nad pozostałymi, wskazać wariantowo kilka klas przy mało różniących się oszacowaniach prawdopodobieństw lub stwierdzić niemożność rozpoznania w wypadku, kiedy wszystkie obliczone prawdopodobieństwa są zbyt małe.

Wybór tej lub innej metody rozpoznawania jest w gruncie rzeczy arbitralny. Prace mające na celu ustalenie, która z metod jest najlepsza, są stosunkowo słabo zaawansowane i trudno w tym zakresie o ścisłe rozstrzygnięcia. Wydaje się możliwe, że w ostatecznym rachunku najkorzystniejsze okażą się metody probabilistyczne, przy czym ich stosowanie musi być poprzedzone obszernymi badaniami zmierzającymi do określenia właściwości statycznych rozkładów cech w poszczególnych klasach.

Rozpoznawanie całych wypowiedzi

Po rozpoznaniu elementów wypowiedzi, wydzielonych w procesie segmentacji, trzeba scalić wyniki uzyskując całościowe rozpoznanie wypowiedzi, a następnie zanalizować rozpoznaną wypowiedź poszukując takich jej cech, które umożliwią zrozumienie sensu wypowiedzi i odpowiednie do jej treści działanie maszyny. Zadania te są bardzo złożone, zwłaszcza w związku z ostatnim z wymienionych problemów, jako że maszynowa analiza treści wypowiedzi wiąże się z licznymi trudnościami. Na szczęście drogi badaczy zajmujących się rozpoznawaniem mowy schodzą się z drogami budowniczych „inteligentnych" translatorów, korygujących błędy źródłowego tekstu programu, a także z pracami nad przetwarzaniem języka naturalnego (w jego pisanej formie) — bardzo intensywnie rozwijanymi w związku z problematyką systemów ekspertowych. Wyniki tych prac można z łatwością adaptować i przystosować do potrzeb automatycznego rozpoznawania mowy, jest to jednak temat na osobny artykuł.

ANNA SYFERT
ANDRZEJ RAZNOWIECKI
JANUSZ ZALEWSKI
(oprac.)
Warszawa

Struktura systemu operacyjnego PC-DOS (2)

Wśród kilku rodzajów przerwań odnoszących się do systemu operacyjnego PC-DOS, przedstawionych w tabeli 1 (por. [3]), najważniejszą rolę odgrywa przerwanie oznaczone INT 21H, służące do wywoływania funkcji systemowych. Poszczególne funkcje wybiera się umieszczając ich kod w rejestrze AH.

Opis tych funkcji sprawia jednak pewien kłopot, ze względu na ich znaczną niejednorodność. Część funkcji systemowych zaprojektowano tak, aby możliwie maksymalnie przypominały wywołania spotykane w systemie CP/M. Miało to ułatwić przenoszenie oprogramowania z mikrokomputerów 8-bitowych na 16-bitowe. Z wywołaniami tymi kojarzy się ściśle funkcje 00H-24H oraz funkcje 25H-2EH dotyczące sterowania systemowego PC-DOS.

Począwszy od wersji drugiej w systemie PC-DOS poczyniono wiele zmian, dodając mu szereg cech charakterystycznych dla systemu Unix i opartego na nim systemu Xenix. Ściśle rzecz biorąc, funkcje o numerach 2FH-38H obejmują zarządzanie pamięcią PC-DOS, a z systemem Xenix kojarzone są funkcje 39H-57H. Funkcje o numerach 58H-62H pojawiły się dopiero w trzeciej wersji systemu PC-DOS. Podział funkcji według tego schematu przedstawiono w tabeli 2.

Do korzystania z wywołań systemowych konieczna jest znajomość opisu plików. W systemie PC-DOS istnieją dwa rodzaje opisu. Są to:

- blok kontrolny pliku, FCB (ang. file control block),
- tzw. uchwyt pliku (ang. file handle).

Istotny dla używania funkcji 00H-24H jest blok kontrolny FCB, natomiast używanie funkcji 39H-62H wymaga korzystania z uchwytów plików, zamiast z bloków FCB. Oba

Tabela 1. Zestawienie niektórych przerwań systemu PC-DOS

Przerwanie	Funkcja
20H	Zakończenie wykonywanego programu
21H	Wywołanie wszystkich funkcji PC—DOS (ang. function dispatcher)
22H	Wektor odpowiadający temu przerwaniu (komórki pamięci 88H—8BH) zawiera adres, do którego wykonuje się skok po zakończeniu wykonywania programu
23H	Wektor odpowiadający temu przerwaniu (komórki pamięci 8CH—8FH) zawiera adres, do którego wykonuje się skok po wprowadzeniu CTRL—C
24H	Wektor odpowiadający temu przerwaniu (komórki pamięci 90H—93H) zawiera adres, do którego wykonuje się skok w wypadku wystąpienia tzw. fatalnego błędu (np. nieczytelna dyskietka)
25H	Odczyt spójnych sektorów dyskietki (odczyt fizyczny)
26H	Zapis spójnych sektorów dyskietki (zapis fizyczny)
27H	Zakończenie aktualnie wykonywanego programu. Program pozostaje w pamięci (np. spooler) i może zostać ponownie wywołany (np. przez przerwanie)
28H—3FH	Przerwania zarezerwowane przez PC-DOS

rodzaje opisu plików stanowią sprzężenie między systemowym mechanizmem obsługi plików a rzeczywistym zbiorem danych zapisanych na dysku.

Blok FCB, oprócz tego, że obsługuje tylko tzw. funkcje uniwersalne 0FH-24H (w wersji 1.10 PC-DOS używa się funkcji 00H-2EH) może być użyty do identyfikowania plików znajdujących się tylko w bieżącym skorowidzu. Uchwyty umożliwiają natomiast identyfikowanie plików w dowolnym skorowidzu za pomocą tzw. funkcji rozszerzonych 39H-62H.

Tabela 2. Podział funkcji systemowych dla przerwania INT 21H

Nr wywołania	Znaczenie
00H—12H	Wprowadzanie i wyprowadzanie charakterystyczne dla CP/M
13H—24H	Zarządzanie plikami charakterystyczne dla CP/M
25H—2EH	Sterowanie systemowe PC—DOS
2FH—38H	Zarządzenie pamięcią PC—DOS
39H—57H	Zarządzenie plikami charakterystyczne dla systemu Xenix
58H—62H	Funkcje trzeciej wersji PC—DOS

BLOK KONTROLNY PLIKU

Blok FCB zawiera informację o strukturze pliku, jego nazwie i długości oraz możliwościach dostępu. Nie zawiera natomiast informacji o położeniu pliku na dyskietce. Ten rodzaj informacji jest zawarty w skorowidzu i tablicy alokacji (por. [2]). Istnienie FCB jest konieczne do wykonania jakiejkolwiek operacji na pliku przy użyciu funkcji z zakresu 0FH-24H.

Tabela 3. Budowa zwykłego lub rozszerzonego bloku FCB

Wyrównanie	Opis
—7H	Charakteryzuje rozszerzony blok FCB za pomocą FFH; rozszerzony FCB jest używany do odwoływania się w skorowidzu do plików ze specjalnym atrybutem
—6H	Zarezerwowane dla PC—DOS (5 bajtów)
—1H	Zawiera atrybut pliku (opis atrybutów przedstawiono w tabeli 4)
00H	Numer dysku: 0 — domyślny dysk 1 — dysk A 2 — dysk B itd.
01H—08H	Nazwa pliku od lewej strony, ewentualnie wypełniona spacjami lub zupełnie pusta; może to być również nazwa urządzenia (np. LPT1 bez dwukropka)
09H—0BH	Typ pliku (rozszerzenie) zapisany od lewej strony, ewentualnie wypełniony spacjami lub zupełnie pusty
0CH—0DH	Wskazuje na bieżący blok danych w obrębie jednego pliku; blok składa się z 128 rekordów danych o długości określonej w następnym polu (ustawiony jest przy użyciu funkcji OFH)
0EH—0FN	Długość logicznego rekordu danych w bajtach; zwykle 128 może być zmieniona po otwarciu pliku
10H—13H	Wielkość pliku w bajtach (ang. file size); bajty 10H—11H zawierają mniej znaczące słowo (16 bitów); zawartość 80H, 06H, 00H, 00H oznacza wielkość równą 1664 bajty
14H—15H	Data ostatniego zapisu w formacie według tabeli 5
16H—17H	Czas dzienny ostatniego zapisu w formacie według tabeli 5
18H—1FH	Zarezerwowane dla PC—DOS
20H	Bieżący rekord danych (0—127) wewnątrz bloku określonego przez wyrównanie OCH—0DH; nie jest inicjalizowany przez funkcję OFH i powinien zostać ustawiony przed odwołaniem programu do pliku sekwencyjnego
21H—24H	Numer (w odniesieniu do początku pliku) bieżącego wybranego rekordu danych przy swobodnym dostępie (ang. random access), przy czym długość użytego rekordu danych określa liczba bajtów użytych dla tego wskaźnika (jeśli długość rekordu danych jest mniejsza od 64 bajtów, to zostają wykorzystane 4 bajty wskaźnika, w przeciwnym przypadku tylko trzy pierwsze); wskaźnik ten musi być również inicjalizowany przed odwołaniem się programu do pliku

Tabela 4. Atrybuty plików w systemie PC-DOS

Kod dwójkowy	Znaczenie
00000000	Nieograniczony dostęp
xxxxxxx1	Plik może być tylko odczytany
xxxxxx1x	Ukryty plik (ang. hidden file); nie może być otwarty za pomocą funkcji OFH
xxxxx1xx	Plik systemowy; nie może być otwarty za pomocą funkcji OFH
xxxx1xxx	Charakteryzuje nazwę logic. nq nośnika (ang. volume ID) w skorowidzu pierwotnym
xxx1xxxx	Charakteryzuje skorowidz podrzędny (ang. subdirectory)
xx1xxxxx	Plik do zarchiwowania (zapisany poleceniem BACKUP); można archiwizować pliki z równoczesnym wyzerowaniem tego bitu
x1xxxxxx	Zarezerwowane dla PC-DOS
1xxxxxxx	Zarezerwowane dla PC-DOS

W tabeli 3 przedstawiono budowę zwykłego (37 bajtów) i rozszerzonego bloku FCB (44 bajty). Rozszerzony blok FCB powstaje wskutek poprzedzenia zwykłego bloku FCB 7-bajtowym prefiksem związanym z atrybutami pliku. W tabelach 4 i 5 przedstawiono sposób budowania atrybutów pliku oraz daty i czasu dziennego jego powstania. Użycie bloków FCB zostanie bliżej omówione w ramach opisów poszczególnych funkcji uniwersalnych. Blok FCB może być umieszczony w obszarze prefiksu segmentu programu FCB1 lub FCB2.

Tabela 5. Format daty i czasu dziennego w bloku FCB (przykładowo, 0AC3H oznacza datę 1985—06—03, a 9760H oznacza godzinę 18:59:00)

Wyrównanie	Bity	Znaczenie
15H—14H	15—9	Dwójkowa reprezentacja roku począwszy od 1980
	8—5	Dwójkowa reprezentacja numeru miesiąca
	4—0	Dwójkowa reprezentacja kolejnego dnia miesiąca
17H—16H	15—11	Dwójkowa reprezentacja godziny
	10—5	Dwójkowa reprezentacja minut
	4—0	Dwójkowa reprezentacja parzystych sekund (do pełnego przedstawienia sekund brakuje jednego bitu)

Należy wspomnieć, że po załadowaniu programu można odczytać z rejestru AX, czy ważna jest nazwa dysku wpisana uprzednio do prefiksu segmentu programu, która zostanie przekazana nieotwartym blokom FCB:

— AL = 00H, nazwa dysku w FCB1 jest ważna,
— AL = 1FH, nazwa dysku w FCB1 jest nieważna.

Rejestr AH dostarcza odpowiednich informacji w odniesieniu do drugiego bloku kontrolnego — FCB2. Nieotwarty blok FCB składa się z prefiksu (jeżeli jest użyty), numeru dysku oraz nazwy i typu pliku — odpowiednio wypełnionych. Otwarty blok FCB ma wypełnione pozostałe pola (przez którąś z funkcji systemowych). Bajty 0-15 (00H-0FH) i 32-36 (20H-24H) bloku FCB muszą być wypełnione przez program użytkowy, natomiast bajty 16-31 (10H-1FH) są wypełnione przez system operacyjny i nie mogą być zmieniane przez użytkownika.

UCHWYT PLIKU

Pojęcie uchwytu pliku wprowadzono w wersji drugiej tego systemu operacyjnego. Mówiąc najprościej, uchwyt pliku jest krótkim sposobem identyfikacji pliku lub urządzenia (16-bitową liczbą udostępnioną w rejestrze AX), analogicznym do numeru kanału w innych systemach operacyjnych.

Znajomość opisu uchwytu pliku nie jest konieczna do używania wywołań systemowych, ponieważ uchwyt pliku jest tworzony przez system operacyjny, a nie przez użytkownika. Do wykonania operacji na pliku konieczne jest

---

[1]) Przypominamy, że ze względów technicznych, zamiast ukośnika (ang. backslash) używamy znaku dzielenia — przyp. red.

Tabela 6. Funkcje wywoływane przez INT 21H

Funkcja	Opis
00H	Zakończenie bieżącego programu
01H	Wprowadzenie znaku z klawiatury z echem na ekran monitora (urządzenie CON)
02H	Wyprowadzenie znaku na ekran monitora (CON)
03H	Wprowadzenie znaku z AUX (sprzęg szeregowy)
04H	Wyprowadzenie znaku do AUX
05H	Wyprowadzenie znaku na drukarkę (PRN)
06H	Wprowadzenie i wyprowadzenie z (na) CON (klawiatura lub ekran)
07H	Wprowadzenie znaku z klawiatury (CON) bez echa i bez rozpoznawania CTRL—C
08H	Analogicznie do 07H, ale z reakcją na CTRL-C
09H	Wyprowadzenie ciągu znaków zakończonych znakiem ,,$''
0AH	Wprowadzenie ciągu znaków (buforowane)
0BH	Kontrola, czy został wprowadzony znak z klawiatury
0CH	Usunięcie wszystkich znaków z bufora klawiatury i wywołanie jednej z funkcji 01H, 06H, 07H, 08H lub 0AH
0DH	Zapisanie wszystkich buforów dysku (jeśli zostały zmienione) na dyskietkę i zerowanie
0EH	Wybranie jednego z kilku dysków
0FH	Otwarcie pliku
10H	Zamknięcie pliku
11H	Poszukiwanie pierwszej nazwy pliku, pasującej do zadanego wzoru (np. *.EXE)
12H	Poszukiwanie kolejnej nazwy pliku
13H	Usuwanie pliku
14H	Odczyt następnego rekordu z pliku sekwencyjnego
15H	Zapis następnego rekordu do pliku sekwencyjnego
16H	Utworzenie nowego pliku
17H	Przemianowanie pliku
18H	Zarezerwowane dla PC-DOS
19H	Sprawdzenie, który dysk został ostatnio wybrany przy użyciu funkcji 0EH
1AH	Zmiana adresu bufora do wprowadzenia-wyprowadzenia dyskowego (DTA, ang. disk transfer address)
1BH	Dostarczenie adresu bajtu identyfikacyjnego tablicy FAT (ang. file allocation table) oraz informacji o pojemności i organizacji wybranego dysku
1CH	Analogicznie do 1BH, można jednak podać numer dysku
1DH—20H	Funkcje zarezerwowane dla PC—DOS
21H	Swobodny dostęp do odczytu pojedynczego rekordu danych danego pliku
22H	Swobodny dostęp do zapisu pojedynczego rekordu danych danego zbioru pliku
23H	Określenie wielkości pliku
24H	Ustawienie numeru rekordu danych dla swobodnego dostępu przy uwzględnieniu parametrów w bloku FCB
25H	Ustawienie wektora przerwań
26H	Utworzenie nowego PSP (prefiksu segmentu programu)
27H	Odczyt bloku danych (kilka rekordów)
28H	Zapis bloku danych (kilka rekordów)
29H	Przeszukiwanie ciągu znaków według zadanej specyfikacji pliku i ewentualne wyodrębnienie jej
2AH	Wyczytanie daty z zegara systemowego
2BH	Ustawienie daty zegara systemowego
2CH	Wczytanie czasu dziennego z zegara systemowego
2DH	Ustawienie czasu dziennego w zegarze
2EH	Zmiana wskaźnika weryfikacji w celu dokonania weryfikacji po zapisie na nośnik
2FH	Pobranie adresu bufora dla operacji dyskowych (DTA)
30H	Pobranie numeru wersji PC-DOS
31H	Zakończenie wykonywanego programu i pozostawienie go w pamięci
32H	Funkcja zarezerwowana dla PC-DOS
33H	Pobranie, ustawienie lub wyzerowanie bitu określającego reakcję na CTRL-C
34H	Funkcja zarezerwowana dla PC-DOS
35H	Pobranie wektora przerwań
36H	Odczytanie informacji o organizacji, obszarze wolnym i całkowitej pojemności nośnika danych
37H	Funkcja zarezerwowana dla PC-DOS
38H	Pobranie daty, czasu dziennego itp., w reprezentacji specyficznej dla danego kraju
39H	Założenie nowego skorowidza (MKDIR)
3AH	Usunięcie pustego skorowidza (RMDIR)
3BH	Zmiana skorowidza (CHDIR)
3CH	Utworzenie nowego pliku
3DH	Otwarcie pliku
3EH	Zamknięcie pliku
3FH	Odczyt z pliku lub urządzenia
40H	Zapis na plik lub na urządzenie
41H	Usunięcie pliku
42H	Przesunięcie wskaźnika zapisu lub odczytu wewnątrz pliku
43H	Zmiana lub pobranie atrybutu pliku (CHMOD)
44H	Pobranie lub zmodyfikowanie informacji o urządzeniu, wprowadzenie-wyprowadzenie specyficzne dla sprzętu (IOCTL, ang. input-output control)
45H	Powielenie uchwytu pliku w celu dostępu do już otwartych plików
46H	Przetworzenie uchwytu jednego pliku na uchwyt drugiego pliku i zamknięcie pierwszego pliku
47H	Określenie, który skorowidz został aktualnie wybrany (przy użyciu funkcji 3BH) dla danego dysku
48H	Rezerwowanie pamięci systemowej RAM
49H	Zwolnienie zarezerwowanej pamięci systemowej
4AH	Zmiana wielkości zarezerwowanego obszaru pamięci systemowej
4BH	Ładowanie programu (np. programu obsługi urządzenia lub nakładki); ładowanie i wykonanie programu
4CH	Zakończenie programu z przekazaniem do wywołującego programu meldunku o błędzie; otwarte pliki zostają zamknięte
4DH	Odczytanie meldunku o błędzie, jaki zostaje przekazany do programu wywołującego przez program wywołany
4EH	Odszukanie pierwszego rozszerzenia nazwy pliku pasującego do zadanego wzorca (np. *.EXE)
4FH	Odszukanie kolejnego rozszerzenia pliku, przy czym plik ostatnio znaleziony przy użyciu funkcji 4EH lub 4FH zostaje użyty w proposzukiwaniu
50H—53H	Zarezerwowane dla PC-DOS
54H	Określenie, czy sprawdzono prawidłowość wykonania zapisu
55H	Zarezerwowane dla PC-DOS
56H	Przemianowanie pliku
57H	Pobranie lub modyfikacja daty i czasu dziennego ostatniego zapisu do pliku
58H	Zarezerwowane dla PC-DOS
59H	Pobranie kodu błędu
5AH	Utworzenie pliku tymczasowego
5BH	Utworzenia nowego pliku, jeśli plik o podanej nazwie nie istnieje
5CH	Blokowanie i odblokowanie dostępu do pliku (ang. lock/unlock)
5EH	Dodatkowe funkcje dla wersji 3.10 i 3.20
5FH	Funkcje związane z przeadresowaniem wejścia-wyjścia (ang. I/O redirection); tylko dla wersji 3.10 i 3.20
60H—61H	Zarezerwowane dla PC-DOS
62H	Pobranie adresu prefiksu segmentu programu PSP

natomiast podanie napisu zawierającego: nazwę dysku, ścieżkę dostępu i nazwę pliku zakończoną bajtem 00H (tzw. format ASCIIZ), np. w języku asemblera [1]).

```
DB "A:/PATH/FILENAME.EXT",0
```

Uchwyt służący do identyfikowania pliku jest udostępniany przez system operacyjny każdorazowo podczas tworzenia i otwierania pliku. Dla nowo tworzonego pliku podaje się również jego atrybuty. Dla otwieranego pliku można również podać tryby dostępu i współdzielenia pliku. Tryb dostępu jest określany według następującej zasady:
0 — plik tylko do odczytu,
1 — plik tylko do zapisu,
2 — plik do odczytu i zapisu.
Tryb współdzielenia określa rodzaj operacji, które można wykonywać współbieżnie na pliku.

Uchwyty są udostępniane w rejestrze AX, a przekazywane do funkcji w rejestrze BX. Atrybuty są przekazywane w rejestrze CX.

Pięć uchwytów jest zdefiniowanych pierwotnie:
0 — standardowe urządzenie wejściowe,
1 — standardowe urządzenie wyjściowe,
2 — standardowe urządzenie do meldunków o błędach,
3 — standardowe urządzenie dodatkowe (AUX),
4 — standardowe urządzenie drukujące.
Liczba uchwytów, których można użyć jednocześnie, jest określona zmienną FILES w pliku CONFIG.SYS (por. [1]). Jeżeli FILES $\geqslant$ 20, to pojedynczy proces może otworzyć co najwyżej 20 plików, przy czym liczba używanych uchwytów musi być mniejsza o trzy uchwyty zdefiniowane pierwotnie (0-2).

* * *

Omawianie wywołań poszczególnych funkcji, przedstawionych w tabeli 6 rozpoczniemy po przedstawieniu — w dwóch najbliższych numerach — zasad pisania programów w języku asemblera.

dokończenie na str. 11

JAN BIELECKI
Instytut Informatyki
Politechnika Warszawska

Język C — definiowanie funkcji ze zmienną liczbą argumentów

Pod koniec lutego pojawiła się w Warszawie wersja 4.0 kompilatora języka C firmy Microsoft. W kompilatorze tym zaimplementowano szereg rozszerzeń języka C zaproponowanych przez ANSI [1]. Należą do nich m.in. typ (void), prototypy funkcji i makrodefinicje do definiowania funkcji ze zmienną liczbą argumentów.

```
main()
{
  void Names(int, ...);

  Names(1,"Jan");
  Names(3,"Ewa","Kaja","Iza");
}

void Names(Count, ...)
  int Count;
{
  char **ArgPtr = (char **)(&Count + 1);

  while(Count--)
    printf("%s\n", *ArgPtr++);
}
```

Wydruk 1. Program zawierający wywołania funkcji ze zmienną liczbą argumentów

Na wydruku 1 przedstawiono przykładową funkcję Names, która może być wywoływana ze zmienną liczbą argumentów. Deklaracją prototypową tej funkcji jest napis:

void Names(int, ...);

Oznacza on, że wynik funkcji jest typu (void), a więc nie istnieje, a ponadto informuje, że funkcja może być wywoływana ze zmienną liczbą argumentów, z których pierwszy (po ewentualnej konwersji) jest typu (int).

```
/*
 * stdarg.h
 *
 * Copyright (C) Microsoft Corporation, 1985, 1986
 *
 */

typedef char *va_list;

#define va_start(ap,v) ap = (va_list)&v + sizeof(v)
#define va_arg(ap,t) ((t*)(ap += sizeof(t)))[-1]
#define va_end(ap) ap = NULL
```

Wydruk 2. Plik standardowy stdarg.h

```
#include <stdio.h>
#include <stdarg.h>

main()
{
  void Names(int, ...);

  Names(1,"Jan");
  Names(3,"Ewa","Kaja","Iza");
}

void Names(Count, ...)
  int Count;
{
  va_list ArgPtr;

  va_start(ArgPtr,Count);

  while(Count--)
    printf("%s\n", va_arg(ArgPtr,char *));

  va_end(ArgPtr);
}
```

Wydruk 3. Program zapisany zgodnie z wymaganiami standardu ANSI

Przytoczony program jest poprawny w sensie implementacji Microsoft (wersja 4.0), ale nie jest programem przenośnym. Poprawność programu wynika z faktu, że w omawianej implementacji argumenty funkcji są odkładane na stosie w kolejności od ostatniego do pierwszego, a usuwaniem argumentów ze stosu zajmuje się funkcja wywołująca. Oznacza to, że wykonanie rozpatrywanego programu powoduje wyprowadzenie (w osobnych wierszach) napisów Jan, Ewa, Kaja, Iza.

W celu uczynienia programu przenośnym należy go zapisać w postaci wymaganej przez ANSI. Program taki, korzystający z pliku nagłówkowego stdarg.h, przytoczonego na wydruku 2, przedstawiono na wydruku 3. Zawiera on włączenie pliku stdarg.h oraz odwołania do makrodefinicji va_start, va_arg i va_end. Występuje w nim ponadto odwołanie do typu va_list.

Program ten spełnia wymagania ANSI, które pokrótce można ująć następująco:

```
#include <stdio.h>
#include <stdarg.h>

main()
{
  void printf(char *, ...);

  printf("Jan%c is %d %s\n", 'B',44,"now");
}

void printf(Format, ...)
  char *Format;
{
  char ch;
  void outdec(int),
       outstr(char *);
  va_list Ptr;

  va_start(Ptr,Format);
  while(ch = *Format++){
    switch(ch){
      case '%' :
        switch(ch = *Format++){
          case 'd' :
            outdec(va_arg(Ptr,int));
            break;
          case 'c' :
            putchar(va_arg(Ptr,int));
            break;
          case 's' :
            outstr(va_arg(Ptr,char *));
        }
        break;
      case '\\' :
        *Format++, ch = '\n';
      default :
        putchar(ch);
    }
  }
  va_end(Ptr);
}

void outdec(dec)
  int dec;
{
  int num;

  if(dec < 0)
    putchar('-'), dec =-dec;
  if(num = dec / 10)
    outdec(num);
  putchar(dec % 10 + '0');
}

void outstr(str)
  char *str;
{
  char ch;

  while(ch = *str++)
    putchar(ch);
}
```

Wydruk 4. Implementacja uproszczonej funkcji printf

- należy zadeklarować pomocniczą zmienną typu va_list;
- należy wywołać makrodefinicję va_start zawierającą odwołanie do zdefiniowanej zmiennej i nazwę ostatniego jawnego parametru funkcji;
- można odwoływać się do makrodefinicji va_arg w celu uzyskiwania kolejnych argumentów funkcji, podając ich typ (po ewentualnej niejawnej konwersji);
- należy wywołać makrodefinicję va_end w celu zakończenia obsługi niejawnych argumentów funkcji.

Na wydruku 4 przedstawiono przykładową przenośną definicję funkcji printf. Dla uproszczenia przyjęto, że ciąg znaków reprezentowany przez pierwszy argument funkcji może zawierać jedynie wzorce %d, %c i %s, zapis znaku nowego wiersza oraz inne znaki widoczne. Wykonanie przytoczonego programu powoduje wyprowadzenie napisu:

JanB is 44 now

LITERATURA

[1] American National Standards Institute: Draft C language standard. New York, 1986
[2] Bielecki J.: Język C — interpretacja standardu, WNT, Warszawa, 1987
[3] Bielecki J.: Wprowadzenie do języka C. WNT (w druku)
[4] Microsoft Corp.: C language compiler, version 4.0. Bellevue (WA), 1986.

Norma ANSI języka C

Twórcy języka C [1] określili go jako „uniwersalny język programowania wyróżniający się oszczędnością wyrażeń, nowoczesnym przepływem sterowania i strukturami danych, a także bogatą listą operatorów". Język C opracowano głównie do wykorzystania w systemie Unix, jednak obecnie jest on coraz częściej stosowany na komputerach osobistych. Duże zainteresowanie językiem i rozwój tworzonego w nim oprogramowania są wystarczającym powodem do opracowania normy. Prace te powierzono Komitetowi Normalizacyjnemu ANSI X3J11.

Po co ta norma?

Przy ciągle zwiększającej się liczbie programujących w języku C, dla zapewnienia wymienności napisanych w nim programów, należy stworzyć uniwersalny zbiór reguł programowania. Problem polega więc na ustanowieniu oficjalnej normy.

Na pierwszy rzut oka wybór wydaje się prosty. Można zezwolić twórcom implementacji (np. AT&T, Whitesmiths lub Lattice) na wypracowanie własnych definicji języka. Ale czy którakolwiek z tych firm mogłaby zagwarantować, że zaimplementowane zamiary i uzupełnienia będą służyły dobru ogólnemu?

Język C ma już rzeczywiście istniejącą normę, tj. podręcznik [1] z oficjalnym przedstawieniem składni i semantyki. Jednakże ze względu na niejasności tekstu, liczne odwołania do Unixa i ostatnie rozszerzenia, nie jest to najlepszy wybór dla piszących kompilatory. W rzeczywistości autorzy kompilatorów mają właśnie zbyt duże możliwości wyboru. Nie jest jasne, czy powinni stosować się do normy [1], czy raczej uwzględnić siódmą wersję Unixa, czy Unix System III, a może najstarszy handlowy kompilator firmy Whitesmiths?

Normalizacja pozwoli na przenoszenie języka do większej liczby środowisk, a ponieważ norma dotyczy tylko języka C, to może nastąpić wyeliminowanie jego zależności od systemów operacyjnych (nigdy nie zlikwiduje się w pełni różnic między nimi) i odejście od jego ścisłego związku z systemem Unix. W rzeczywistości dwie trzecie firm reprezentowanych w Komitecie Normalizacyjnym oferuje język C dla środowisk innych niż Unix.

Norma ANSI pozwoli na prawidłowy rozwój języka z pominięciem przeszkód, które napotkały inne języki programowania. Pozwoli również uniknąć sytuacji, w której „dojrzewanie" języka odbywałoby się pod kontrolą pojedynczego twórcy kompilatora lub systemu operacyjnego.

Biuletyn informacyjny

W marcu 1984 roku na spotkaniu Komitetu Normalizacyjnego X3J11 przegłosowano opublikowanie aktualnego wówczas projektu normy ANSI języka C jako Biuletynu Informacyjnego [2].

Biuletyn Informacyjny jest tylko jednym z kroków na drodze do normalizacji języka C. Jest to krok początkowy, ale niezwykle ważny, ponieważ producenci kompilatorów, implementując w odpowiednim środowisku część lub całość projektu normy, uzyskują możliwość sprawdzenia efektów pracy kompilatora. Próbne zaimplementowanie nowych cech opisanych w Biuletynie zaznajomi użytkowników z zaproponowanymi zmianami i pozwoli na określanie niedostatków obecnej wersji języka.

W artykule omówiono niektóre uzupełnienia i zmiany w języku C zaproponowane w Biuletynie Informacyjnym [2].

Rozszerzenia typu unsigned

W definicji [1] słowo zastrzeżone **unsigned** oznacza typ całkowitoliczbowy **(int)**. W wielu późniejszych kompilatorach słowo zastrzeżone **unsigned** rozszerzono o znaczenie modyfikatora. W obecnym projekcie do zwykle wykorzystywanego znaczenia typu **unsigned** dodano następujące. **unsigned char, unsigned short, unsigned int, unsigned long**. Tę modyfikację uważa się za niewielkie rozszerzenie w stosunku do [1], nie wpływające na istniejące programy.

Niepowtarzalne nazwy składowych

Proponowana norma dopuszcza stosowanie niepowtarzalnych nazw składowych struktur i unii. Programiści mogą posługiwać się nazwą składowej, np. Name, w obrębie wielu różnych struktur bez konieczności zapewnienia równych przemieszczeń (ang. displacement) w każdej strukturze. Wszystkie unie i struktury mają odpowiednią przestrzeń na nazwy, a zadaniem kompilatora jest sprawdzenie ważności nazw poszczególnych składowych. Może to spowodować kłopoty ze starymi programami, w których zdefiniowano struktury i unie, a następnie wykorzystano nazwy składowych jako wartości ogólne.

Przekazywanie i przypisywanie struktur

W Biuletynie Informacyjnym zawarto propozycję przekazywania i przypisywania struktur (w postaci wzorowanej na Unixie System V). Oznacza to możliwość przypisania jednej struktury innej strukturze, przekazania całej struktury jako argumentu na stos i utworzenia funkcji udostępniającej wartość będącą strukturą. Te cechy pozwalają funkcji operować kopią struktury pierwotnej, a po wprowadzeniu zmian wykonać przypisanie i w ten sposób uaktualnić strukturę pierwotną.

Sygnał dźwiękowy

Umieszczając w napisie wyprowadzanym literał „a poprzedzone ukośnikiem" można przesyłać siódmy znak ASCII (sygnał dźwiękowy, ang. bell) jako sygnał ostrzegawczy. Rozważono również dodatkowo możliwości uwzględnienia właściwości terminali, lecz liczba rozwiązań sprzętowych i konieczność ograniczenia działalności komitetu tylko do języka sprawiła, że poprzestano na tej jednej propozycji. Jest to zresztą jedyne rozwiązanie związane ze sprzętem zawarte w Biuletynie.

Międzynarodowe operatory trójznakowe

W języku C wykorzystuje się zbiór znaków ASCII, co jest bardzo wygodne dla programistów amerykańskich. Niektóre znaki używane jako operatory w języku C są rozszerzeniem alfabetów wielu języków etnicznych, na przykład niemieckiego, francuskiego, hiszpańskiego itd.

Komitet podjął problem brakujących znaków ASCII, definiując w Biuletynie kilka ciągów trójznakowych działających jako równoważniki operatorów. Ułatwi to szersze międzynarodowe wykorzystanie języka C. Ciągi trójznakowe pozwalają definiować znaki nie będące częścią zbioru znaków (ang. invariant code set) ISO 646, podzbioru 7-bitowego kodu ASCII (rys.). Dodatkowe zdefiniowanie znaku zapytania jako „? poprzedzany ukośnikiem" pozwala na jednoczesne drukowanie tych znaków i ciągów trójznakowych.

Trójznak	??=	??(	??/	??)	??'	??<	??!	??>	??-
Znak definiowany	#	[	\	]	^	{	|	}	~

Operatory definiowane przez ciągi trójznakowe

Zmiany słów zastrzeżonych

Propozycja ANSI zawiera pięć nowych słów zastrzeżonych. Nowe słowa zastrzeżone dodano z rozwagą, ponieważ programy dostosowywane do normy ANSI musiałyby zostać zmienione, gdyby określone słowo zastrzeżone spełniało w nich inną funkcję. Pięć nowych słów zastrzeżonych to:

qvoid, enum, const, signed i **volatile.**

Słowo zastrzeżone **void** przeniesiono do projektu normy z Unixa System V, aby umożliwić programiście definiowanie funkcji nie udostępniającej żadnej wartości. Na przykład, deklaracja:

```
extern void exit ( ) ;
```

przekazuje kompilatorowi, że nie oczekuje się żadnej znaczącej wartości udostępnianej przez funkcję. Pozwala to kompilatorowi zasygnalizować następujące wyrażenie jako błąd:

```
a = b + exit ( ) ;
```

Typ **enum** został włączony do projektu normy w postaci używanej w Unixie System V. Typ **enum** jest podobny do typu wyliczeniowego w Pascalu, ale implementowany zgodnie z duchem języka C.

Słowo zastrzeżone **const** wykorzystują już wewnętrzne kompilatory firmy AT&T. Jest to specyfikator typu, definiujący obiekt niemodyfikowany albo rezydujący w pamięci ROM. Obiekty typu **const** nie mogą być przypisywane, inkrementowane lub dekrementowane. Typ **const** ma dwie główne zalety; umożliwia identyfikowanie danych, które nie mogą być zmienione ani w module, ani w całym programie, oraz zapewnia rozpoznawanie danych umieszczonych w pamięci ROM jako specyficznej grupy. Po rozpoznaniu typu danych jako **const,** dane są sprawdzane przez kompilator w celu zasygnalizowania błędu, jeśli programista próbuje je zmienić.

Język C według definicji [1] pozwala implementatorom kompilatorów wybrać, czy typ **char** ma mieć wartość ze znakiem czy bez. Pozwala to uzyskać największą możliwą szybkość tworzenia kodu dla elementów typu **char.** Oznacza to, że dodanie do **char** modyfikatora typu **unsigned** powoduje otrzymanie wartości bez znaku. Ale co począć z wartościami ze znakiem? Zawarte w projekcie normy słowo **signed** pozwala tworzyć wartości ze znakiem.

Dodanie do języka C słowa zastrzeżonego **signed** eliminuje również problem polegający na tym, że przy naturalnym typie **char** programista nie może otrzymać wartości **char** ze znakiem z zakresu [—128, 127]. Po dodaniu tego słowa programista uzyskuje pełną kontrolę nad tym, czy dane typu **char** są ze znakiem, bez znaku lub wybrane przez kompilator.

Modyfikator typu **signed** może być użyty wszędzie tam, gdzie jest dozwolony typ **unsigned.** Pozwala to programiście w sposób szczególny deklarować zmienne, tym samym zwiększając czytelność i wydajność programu.

Słowo zastrzeżone **volatile** może zostać wykorzystane do opisu środowiska kompilatora. Obecnie, po tym jak kompilator ładuje jakąś wartość do rejestru (jeśli nie jest używana dalej w programie) zakłada się, że jest ona taka sama jak zawartość komórki pamięci wykorzystana do załadowania rejestru. Taka sytuacja kryje w sobie kilka pułapek. Na przykład, co się dzieje, jeśli występuje wspólne adresowanie pamięci i modułów we-wy, a rozważana komórka jest portem komunikacyjnym? Podobnie, co się dzieje, gdy komórka znajduje się w obszarze pamięci wspólnej, modyfikowanej przez inny proces? Wartość przesłana do rejestru może zostać zmieniona po pewnym czasie lub po zajściu kilku zdarzeń.

Istnieje więc pewna niezgodność z podstawowym wymaganiem normy [1], polegającym na tym, że wartość raz załadowana do rejestru nie musi być ponownie ładowana, jeśli nie zaszło żadne zdarzenie zmieniające zawartość odpowiedniej komórki pamięci. Podstawowy model pamięci i kompilatora według [1] nie pozwala programiście opisać warunku, w którym zawartość komórki pamięci zostanie zmieniona na skutek działań zewnętrznych. Jest to bardziej oczywiste w środowisku wieloprogramowym, takim jak Unix. Jeśli istnieje obszar wspólnej pamięci, to zawartość komórki, na której działa kompilator może się zmienić od czasu jej ostatniego użycia.

Główna zaleta związana z wykorzystaniem słowa zastrzeżonego **volatile** polega na poprawieniu jakości optymalizacji wykonywanej przez kompilator; programista może lepiej opisać właściwe środowisko kompilatora. Pozwala to zoptymalizować operowanie zmiennymi i właściwie manipulować obszarami danych specjalnych w różnych komputerach.

W projekcie normy pominięto słowo zastrzeżone **entry** jako nie wykorzystywane w definicji [1].

Typy long i unsigned int

W języku C przyjęto, że stała (np. 123) jest typu **int.** Jest to więc wartość o znanym module ze znakiem, która może być użyta w wyrażeniach zawierających inne dane typu **int.** Kiedy stała pojawia się w wyrażeniu zawierającym typy **long,** można na jej końcu umieścić literę „l" oznaczającą **long** (123l i 123L są przykładami wartości typu **long int**).

Nie działa to dobrze, jeśli stałe określają adresy. Adresy są interpretowane jako liczby bez znaku. W projekcie normy dopuszcza się specyfikację stałych z dodatkową, kończącą literą „u", oznaczającą wartość bez znaku (na przykład: 123u i 123U). Taki sposób oznaczania stałych bez znaku pozwala programiście adresować pamięć i wykonywać inne działania w arytmetyce bezznakowej.

Długości typów short, int i long

Kompilatory oparte na definicji [1] traktują słowa zastrzeżone **short** i **long** jako synonimy **int.** W komputerach 16- i 32-bitowych specyfikatory **short** i **int** oznaczają zwykle 16 bitów, natomiast **long** — 32 bity. W wersji zaproponowanej w projekcie [2] wszystkie trzy typy mogą mieć różne długości. Na przykład, duże komputery mogłyby mieć typ **short** o długości 16 bitów, **int** — 32 bity i **long** — 64 bity. Zezwolenie na różne długości typów może okazać się pożyteczne tylko dla niektórych dużych komputerów. Jednakże, zachowanie trzech różnych typów nie jest koniecznością i określenie ich wielkości może zależeć od implementacji.

Rozszerzenia dla programistów Fortranu

Odkąd język C stał się tak popularny, wielu programistów Fortranu rozważa przestawienie się na programowanie w tym języku. Żeby zachęcić do takich zmian, Komitet X3J11 wprowadził trzy rozszerzenia, przybliżające język C do potrzeb użytkowników Fortranu. Rozszerzenia te zawierają:

- obliczanie wyrażeń w arytmetyce zmiennoprzecinkowej,
- możliwość narzucenia kolejności wyznaczania wartości wyrażeń,
- stworzenie nowego typu **float,** dłuższego od **double.**

Jednocześnie Komitet Normalizacyjny zadbał o to, aby nie trzeba było w związku z tym zmieniać istniejących programów.

Składniki wyrażeń mogą być definiowane na poziomie typu **float**. Pozwala to wykonywać obliczenia w arytmetyce zmiennoprzecinkowej, zamiast podwójnej precyzji, co z kolei umożliwia kompromis między szybkością i dokładnością wykonywanych działań.

W celu umożliwienia programiście ustalania kolejności obliczania składowych wyrażenia dodano operator jednoargumentowy „+". Na przykład, jeśli należy obliczyć wartość wyrażenia A + (B + C), to kompilator najpierw doda A i B, a potem C. Jedyny sposób na to, aby B + C było obliczone w pierwszej kolejności, polega na użyciu zmiennej tymczasowej:
temp = B + C;
wynik = temp + A;
Niestety, wymaga to zadeklarowania dodatkowej zmiennej i może zmienić styl kodowania. Projekt normy pozwala powyższe wyrażenie zapisać jako A + + (B + C). Dodatkowy znak dodawania informuje kompilator o tym, że najpierw należy obliczyć wyrażenie w nawiasach.

Trzecia cecha przyjazna dla użytkowników Fortranu, to istnienie nowego typu **float**, dłuższego od **double**. Jest to typ nazwany **long double**. Może on być taki sam jak **double**, ale również o większej precyzji. Ten typ pozwala twórcom funkcji matematycznych stosować większą precyzję niż normalnie. Firma Intel wykorzystuje ten sam pomysł w koprocesorze 8087; wewnętrzne obliczenia wykonuje się na 80 bitach, a komunikacja odbywa się przy użyciu 64 bitów.

Preprocesor

Preprocesor języka C jest dość dziwnym tworem, którego działanie zależy od twórców kompilatora i ich inwencji w dokładnym określeniu fazy przetwarzania wstępnego.

Projekt normy zawiera dwa nowe operatory preprocesora: # (znak numeru, krzyżyk, ang. sharp) i ## (podwójny znak numeru, ang. double sharp). Te dwa rozszerzenia umożliwiają wykonywanie operacji poprzednio dostępnych w niektórych (ale nie wszystkich) kompilatorach języka C. Obydwa operatory działają na napisy i makrodefinicje. Wielu programistów używa makrodefinicji w celu stworzenia wartości później wykorzystywanej jako napis lub nowy operator (na przykład nazwa zmiennej). Na pierwszy rzut oka wygląda to dość prosto, ale istnieje jeszcze wiele problemów nie rozwiązanych we współczesnym programowaniu. Najlepiej opisuje to przykład zawarty w Biuletynie:
define debug(s,t) printf("x" #s "=%d,x" #t "=%s", x##s, x##t)

debug (1,2)

co oznacza:

printf ("x" "1" "=%d,x" "2" "=%s", x1, x2)

W wyniku konkatenacji sąsiednich stałych tekstowych otrzymuje się:

printf ("x1=%d, x2=%s", x1, x2)

Jeśli po znaku # występują formalne parametry makrodefinicji, to zostają zastąpione przez odpowiednie parametry aktualne i ujęte w cudzysłowy. Przedstawiono to wyżej, gdzie #s zastąpiono przez "1". Umożliwia to łatwe budowanie literałów napisowych w makrodefinicji. Znak ## jest operatorem konkatenacji. Po wykonaniu zamian określonych makrodefinicją, znaki ## są usuwane, a znaki poprzedzające i znajdujące się za operatorem ## ulegają konkatenacji.

Długość napisów

Powyższe przykłady zawierają inną propozycję zmiany obecnej praktyki językowej. Propozycja ANSI zezwala na to, aby napisy były dłuższe niż używane obecnie. Programista może utworzyć napis umieszczony w kilku sąsiednich wierszach programu, logicznie sklejonych. Oznacza to równoważność dwóch następujących przykładów:

```
printf ("this is a single string with the numbers one, two
        and three/");[1])
```

```
prinft ("this is a"
                "single string"
                "with the"
                "numbers"
                "one"
                "two and three" ) ;
```

Zmiany zdefiniowanych pierwotnie makrowartości

Zaproponowano zmianę zdefiniowanych pierwotnie makrowartości __LINE__ i __FILE__. Wartość __LINE__ zdefiniowano jako numer aktualnego wiersza, a __FILE jako nazwę aktualnego pliku źródłowego, lub pliku **include**. Powyższych makrowartości można używać przy drukowaniu informacji dotyczących aktualnego pliku i numeru wiersza.

Dyrektywa #pragma

Dyrektywa preprocesora #**pragma** została włączona do proponowanej normy po to, aby twórcy kompilatorów mogli wprowadzać do nich dodatkowe informacje, bez popotrzeby dodawania nowych słów zastrzeżonych. Makrodefinicja #**pragma** ma tak ogólny mechanizm działania, że dowolne informacje o środowisku, właściwościach i rozszerzeniach są rzeczywiście przenośne między kompilatorami. Informacja występująca w linii tej dyrektywy, nie rozpoznawana przez kompilator lub nie określona w jego środowisku jest ignorowana, nie powodując wysłania komunikatu o błędzie.

Prototypy funkcji

Jest to rozszerzenie składni dotychczasowej deklaracji **extern function** ( ). Nowa składnia dopuszcza stosowanie poprzedniej, ale jednocześnie programiści i twórcy bibliotek mają możliwość podawania dodatkowych informacji o procesie kompilacji.

Zwykłe problemy występujące przy wywoływaniu funkcji można zademonstrować na przykładzie funkcji **lseek**. Funkcja ta ma trzy argumenty i udostępnia wartość typu **long**. Zwykle ta funkcja jest dana w postaci:

extern long lseek ( ) ;

Taka postać jest dopuszczalna i w dalszym ciągu będzie działać w istniejących programach. Propozycja ANSI pozwala jednak dodać prototyp funkcji:

extern long lseek (**int** file, **long** position, **int** mode);

Prototyp funkcji przekazuje kompilatorowi informacje, które może wykorzystać programista. Pierwszy rodzaj informacji opisuje argumenty funkcji. W tym wypadku, **lseek** ma trzy i tylko trzy argumenty. Wywołanie funkcji **lseek** z inną liczbą argumentów, kompilator potraktuje jako błąd. Podobnie, gdy drugi argument nie jest typu **long**, to prototyp funkcji przeprowadza konwersję wartości na wymagany typ. Konwersja jest wykonana w taki sposób, jak gdyby przypisanie było określone za pomocą znaku równości.

Taka kontrola typów przy wywoływaniu funkcji stanowi dużą część programu usługowego **lint** w systemie operacyjnym Unix. Jednakże kontrola według prototypu funkcji jest silniejszym narzędziem, ponieważ wykonuje się ją podczas kompilacji, a nie oddzielnie w jakiś czas po skompilowaniu programu. Ponadto, ponieważ wszystkie pliki nagłówkowe biblioteki standardowej zawierają prototypy funkcji bibliotecznych, programista jest zawiadamiany w trakcie kompilacji o błędnej liczbie argumentów i błędnych typach. Dotychczas te informacje otrzymywało się później, podczas uruchamiania programu lub po wykonaniu programu usługowego **lint**.

W celu zapewnienia czytelności, w prototypach funkcji mogą występować nazwy. Osoba dokumentująca funkcję może umieścić w prototypie funkcji nazwę opisującą jego wykorzystanie. Wiele kompilatorów wyposażonych w prototypy funkcji może udostępniać w procesie kompilacji plik definicji prototypów. Dzięki temu automatyczne wytwarzanie prototypów funkcji można zawrzeć w innych modułach, które używają funkcji.

Programista posługujący się prototypami funkcji nie jest zmuszony do tego, aby wszystkie argumenty były automatycznie rozszerzane do typu **double** lub **int**. Obecność prototypu w definicji i w pliku wykorzystującym funkcję pozwala określić typ argumentu jako **float**, **char** lub nawet **register**. Umożliwia to lepszą optymalizację kompilatorów

[1]) Przypominamy, że ze względów technicznych zamiast ukośnika (ang. backslash) używamy w INFORMATYCE znaku dzielenia — przyp. red.

i sprawowanie większej kontroli nad przepływem informacji do podstawowej jednostki języka — funkcji.

* * *

Przez ostatnie dwa i pół roku Komitet Normalizacyjny rozwiązywał różnorodne problemy. Przedstawiano różne warianty rozwiązań, przygotowywano sprawozdania zawierające ich analizy, podejmowano decyzje i dokumentowano je w Biuletynie Informacyjnym. W toku tych prac rozwiązano takie kwestie:

- czy należy rozszerzać funkcjonalność typu **enum**;
- jaka powinna być długość elementów typu **external**;
- czy nazwy elementów typu **external** mogą być zapisywane w sposób dowolny (bez rozróżniania małych i dużych liter), czy też literami jednego rodzaju;
- w jaki sposób należy zmieniać lub rozszerzać wartości podczas obliczania wyrażeń zawierających wiele typów;
- jaką kontrolę musi wykonać kompilator, aby wykryć próbę zmiany wartości zadeklarowanej jako **const**;
- czy elementy typu **external** muszą mieć jedną, czy też wiele definicji, gdy są inicjalizowane na tę samą wartość.

Przyjęte rozwiązania stanowią próbę uczynienia z języka C narzędzia użytecznego dla jak największej liczby użytkowników.

Oprac. M. KUC

LITERATURA

[1] Kernighan B. W., Ritchie D. M.: The C Programming Language. Prentice-Hall, Englewood Cliffs (NJ), 1978

[2] CBEMA: The C Language Information Bulletin. Computer and Business Equipment Manufacturers Association, Washington (DC), 1984

[3] Winiewski J.: Język programowania C. Informatyka, nr 4—6, 1985

[4] Bielecki J.: Błędy implementacji w kompilatorze języka C firmy Supersoft. Informatyka, nr 1, 1986

[5] Bielecki J.: Zaawansowane konstrukcje języka C. Informatyka, nr 2—3, 1986.

Zakład Usług Informatycznych
R. Brykajło
ul. J. Bojki 6/22, 30-612 Kraków
tel. 34-50-93

poleca

ODRA-1305

- system redagowania i uruchamiania z monitorów lokalnych zadań George-2,
- interfej ODRA — AMSTRAD 6128,
- pomoc przy wdrożeniu systemów George 2 i 3.

MIKROKOMPUTERY

- system kosztorysowania,
- system płac,
- procedury dostępu do plików dBase z poziomu Pascala Turbo i MT+,
- procedury grafiki dla AMSTRAD 6128 w Pascalu Turbo i MT+.

ORAZ

- usługi w zakresie projektowania i programowania systemów przetwarzania danych.

EO/901/87

ZEKOM
ZAKŁAD ELEKTRONIKI KOMPUTEROWEJ
ul. Makowa 8, 91-480 Łódź tel. 34 30 49

oferuje użytkownikom systemów MERA-9150 i REDIFON

TERMINAL EKRANOWY
MR 1240

będący funkcjonalnym odpowiednikiem monitora MERA-795.
Terminal MR 1240 przeznaczony jest do wprowadzania danych do systemu minikomputerowego MERA-9150

JEŚLI INTERESUJE PAŃSTWA:

- profesjonalny sprzęt o wysokiej jakości, niezawodny w eksploatacji
- sprawny serwis
- krótkie terminy dostaw, lub dostawy natychmiastowe

TO WYROBY ZEKOMU SĄ DO PAŃSTWA DYSPOZYCJI

dla użytkowników systemów MERA-9150 i REDIFON kompatybilny z MERA-7951

EO/307/87

Drugie wejście Codda

Kiedy w roku 1970 E. F. Codd opublikował na łamach „Communications of the ACM" artykuł pt. „A relation model of data for large shared data banks", mało kto przypuszczał, że koncepcja ta stworzy podwaliny nowoczesnej teorii i praktyki systemów zarządzania bazami danych. O dzisiejszym, ciągle aktualnym znaczeniu prac Codda najlepiej świadczą tysiące cytowań oraz fakt, że wśród komercyjnych, mikrokomputerowych systemów zarządzania bazami danych szczególną popularnością cieszą się systemy o rodowodzie relacyjnym, na przykład dBASE II, III i III plus, MicroRIM i inne.

Ostatnio, niemal równocześnie, dwa czołowe wydawnictwa europejskie opublikowały monografie relacyjnych baz danych. Wydawnictwo North-Holland wydało pod koniec roku 1985 książkę C. Delobela i M. Adiby pt. „Relational Database Systems" (ISBN: 0444877185), a Springer-Verlag w roku 1986 — książkę znanego już w Polsce autora, Suada Alagića, zatytułowaną „Relational Database Technology" (ISBN: 038796276X). Obie książki, mimo niemal identycznego zakresu poruszanych zagadnień, różnią się dość wyraźnie.

Książka Delobela i Adiby jest tłumaczeniem wydanego w roku 1983 we Francji tekstu „Bases de Donnees et Systémes Relationnels". Do niewątpliwych zalet książki należy jej przejrzysty układ. Część I jest poświęcona modelom danych; oprócz szczegółowej prezentacji modelu relacyjnego scharakteryzowano także model hierarchiczny i sieciowy oraz pokrewny model binarny. W części II przedstawiono zagadnienia dotyczące implementacji (SEQUEL, Query-By-Example), architektury systemów, pamięci relacyjnej, optymalizacji zapytań, a także problem integralności. W części III omawiana jest normalizacja. Jest to układ uzasadniony, ponieważ na ogół mamy do czynienia ze swoistym kultem normalizacji, którego znaczenie praktyczne dla konkretnych instalacji relacyjnych baz danych było znacznie mniejsze niż to, które podkreślali autorzy kolejnych n-tych postaci normalnych.

Książka Delobela i Adiby nie jest pozbawiona wad. Wydaje się, że autorzy postąpiliby lepiej pisząc ponownie książkę w języku angielskim, aniżeli powierzając osobie trzeciej tłumaczenie z języka francuskiego. Niewątpliwie zyskałby na tym tekst, w obecnej postaci w wielu miejscach niejasny, dwuznaczny czy wręcz trudno zrozumiały. Drugi zarzut, natury znacznie poważniejszej, dotyczy literatury, która nie wykracza poza rok 1981. Zapewne dlatego omawia się dawny SEQUEL, a nie aktualny SQL. Szkoda, że przy okazji tłumaczenia nie wprowadzono stosownych uzupełnień. Dobrej jakości, sztywna i efektowna plastycznie okładka spina książkę wydrukowaną w technice foto-offsetu, z matryc dostarczonych przez autorów. Niestety, spośród 451 stron ok. 70 nie wytrzymuje kryterium przyzwoitej czytelności i dziwić się należy, że wydawnictwo nie wyegzekwowało od autorów dostarczenia matryc (maszynopisów) odpowiedniej jakości.

Książka Suada Alagića jest zatytułowana w niebanalny, by nie rzec wyszukany sposób i ma bardzo przejrzysty układ. W części I przedstawiono podstawy relacyjnego modelu danych. W części II omówiono projektowanie logiczne (normalizacja do piątej postaci normalnej — ang. projection/join, ale bez postaci normalnej DK — ang. domains and keys), z uwzględnieniem rozszerzonego modelu RM/T. W części III przedstawiono tzw. projektowanie strukturalne, przy czym szczególny nacisk położono na dekompozycję i optymalizację zapytań. Część IV jest poświęcona problematyce systemów o charakterze wielodostępnym (w oryginale — multiuser environment), tj. integralności, współbieżnemu wykonywaniu transakcji i odtwarzaniu integralnego systemu bazy danych. W części V zostały omówione podstawowe zagadnienia rozproszonych baz danych — architektura, kryteria rozpraszania, rozproszone przetwarzanie zapytań i rozproszona aktualizacja. Do niewątpliwych zalet książki Alagića należy bardzo przejrzyste przedstawienie pełnego zestawu algorytmów operacji algebraicznych w pascalopodobnej notacji, merytorycznie nienaganne ilustracje i diagramy oraz poważne uwzględnienie nowego kierunku, jakim są rozproszone systemy relacyjnych baz danych. Książka, choć nie pozbawiona paru błędów natury korektorskiej (np. niektóre definicje w podrozdziale 3.3), jest zredagowana i wydana bardzo starannie. Umiarkowana objętość (259 stron), nowa, sięgająca roku 1985 literatura, liczne ćwiczenia i testy sprawiają, że książka ta może stanowić doskonałą podstawę systematycznego wykładu na poziomie uniwersyteckim.

Nie kwestionując walorów książki Delobela i Adiby, czytelnikowi pragnącemu gruntownie poznać aktualny stan wiedzy na temat relacyjnych baz danych zdecydowanie polecam książkę Suada Alagića.

PIOTR J. JASIŃSKI

WARUNKI PRENUMERATY NA 1988 ROK

Prenumeratorzy zbiorowi — jednostki gospodarki uspołecznionej, instytucje i organizacje społeczne zamawiają prenumeratę dokonując wpłaty wyłącznie na blankiecie „wpłata-zamówienie" (jest to „polecenie przelewu" rozszerzone dla potrzeb Wydawnictwa o część dotyczącą zamówienia). Blankiety te będą dostarczane dotychczasowym prenumeratorom przez Zakład Kolportażu. Nowi prenumeratorzy otrzymają je po zgłoszeniu zapotrzebowania (pisemnie lub telefonicznie) w Zakładzie Kolportażu.

Prenumeratorzy indywidualni — osoby fizyczne zamawiają prenumeratę dokonując wpłaty w UPT lub NBP na blankiecie NBP. Na odwrocie wszystkich odcinków blankietu należy wpisać tytuł czasopisma, okres prenumeraty, liczbę zamawianych egzemplarzy oraz wartość wpłaty. Wpłacać należy na konto: NBP III Oddział Warszawa 1036-7490-139-11.

Prenumerata ulgowa — przysługuje wyłącznie osobom fizycznym — członkom SNT, studentom i uczniom szkół zawodowych. Warunkiem prenumeraty ulgowej jest poświadczenie blankietu wpłaty (przed jej dokonaniem) na wszystkich odcinkach pieczęcią Koła SNT, wyższej uczelni lub szkoły. Sposób zamawiania prenumeraty ulgowej jest taki sam jak prenumeraty indywidualnej. W prenumeracie ulgowej można zamówić tylko po 1 egzemplarzu każdego czasopisma.

Uwaga! Miesięcznik „Aura" może być zamawiany w prenumeracie ulgowej również przez uczniów szkół ogólnokształcących.

Prenumerata ze zleceniem wysyłki za granicę — zamawia się tak jak prenumeratę indywidualną. Dodatkowo należy podać na blankiecie wpłaty nazwisko i dokładny adres odbiorcy. Cena prenumeraty ze zleceniem wysyłki za granicę jest dwukrotnie wyższa.

Wpłaty na prenumeratę przyjmowane są w terminach:
— do 10 listopada na każdy kwartał, I i II półrocze oraz cały rok następny,
— do 28 lutego na II, III i IV kwartał oraz II półrocze,
— do 31 maja na III i IV kwartał oraz II półrocze,
— do 31 sierpnia na IV kwartał.
Zmiany w prenumeracie można zgłaszać pisemnie tylko w wyżej wymienionych terminach.

Informacji o prenumeracie udziela — Zakład Kolportażu Wydawnictwa NOT SIGMA (ul. Bartycka 20, 00-716 Warszawa), skr. poczt. 1004, 00-950 Warszawa, tel. 40-00-21 wew. 248, 249, 293, 297, 299 lub 40-30-83 i 40-35-89.

Egzemplarze archiwalne czasopism — można nabyć za gotówkę w Klubie Prasy Technicznej, Warszawa ul. Mazowiecka 12 (tel. 27-43-65), lub zamówić pisemnie. Zamówienia na egzemplarze archiwalne czasopism przyjmuje: Zakład Kolportażu, Dział Handlowy, 00-950 Warszawa, skr. poczt. 1004 (tel. 40-37-31), na rachunek dla instytucji lub za zaliczeniem pocztowym dla osób fizycznych.

Cena prenumeraty wg cennika na 1988 rok					
kwartalna		półroczna		roczna	
normalna	ulgowa	normalna	ulgowa	normalna	ulgowa
600,—	150,—	1200,—	300,—	2400,—	600,—

Cena egzemplarza 200 zł (50 zł — cena ulgowa).

Programowanie w Adzie rozproszonych systemów docelowych

W artykule przedstawiono cztery podejścia do programowania w Adzie systemów rozproszonych. W pierwszej metodzie wykorzystuje się zatwierdzony kompilator Ady do wytwarzania kodu wynikowego dla rozproszonych obiektów docelowych. Drugie podejście ogranicza rozwiązania programowe, tak że jedyną dopuszczalną formą komunikacji między słabo powiązanymi procesorami jest spotkanie zadań. Trzecia metoda polega na podzieleniu przed kompilacją kodu źródłowego Ady na oddzielne bloki plików źródłowych dla każdego procesora docelowego. Czwarte podejście wykorzystuje nieco bardziej tradycyjną metodę. Obejmuje ona projektowanie i konstruowanie oddzielnych programów źródłowych dla każdego procesora systemu rozproszonego. Natomiast oprogramowanie sprzężenia międzyprocesorowego jest w tej metodzie kodowane niezależnie dla każdego zastosowania.

Ważne jest, że każde z tych podejść umożliwia wytwarzanie oprogramowania rozproszonego bez modyfikacji samego języka. Optymalna metoda powinna stawiać jak najmniejsze ograniczenia wobec projektu oprogramowania.

KOMPILATORY DLA ROZPROSZONYCH OBIEKTÓW DOCELOWYCH

Pierwsze podejście polega na użyciu kompilatora wytwarzającego kod dla obiektów rozproszonych. W tej metodzie oprogramowanie jest napisane w Adzie jako pojedynczy program aplikacyjny. Pragmy informują kompilator, jakie procesory znajdują się w zestawie i dokąd mają być przydzielone różne segmenty kodu. Na tej podstawie kompilator wytwarza dla każdego procesora w systemie oddzielne pliki kodu wynikowego.

Jeśli alokacja programu nie jest jasno określona przez pragmy, a zostanie wykonana przez kompilator, to może powstać kilka problemów. Przede wszystkim, projektant nie będzie świadomy sposobu podziału oprogramowania przez kompilator, gdyż kod źródłowy nie odzwierciedla wynikowej architektury oprogramowania. Może to radykalnie wpłynąć na sprawność całego systemu; jako że kompilator nie musi wykonywać najbardziej efektywnego podziału dla konfiguracji docelowej.

Największy problem stanowi jednak implementacja kompilatora dla rozproszonych obiektów docelowych. Jeśli procesory nie współdzielą pamięci, to w celu zsynchronizowania procesorów musi istnieć rozproszone jądro wykonawcze (ang. run-time kernel). Aby kompilator obsługiwał niejednorodne obiekty docelowe, musi rozpoznawać wszystkie różnice reprezentacji danych wewnętrznych; wymaga też wielu generatorów kodu.

Te kwestie w znacznym stopniu komplikują zaprojektowanie takiego kompilatora.

KONCEPCJA ROZPROSZONEGO ŚRODOWISKA PROGRAMOWEGO

W celu pokonania problemów występujących w programowaniu obiektów rozproszonych, można nałożyć na program pewne ograniczenia projektowe. Jedna z propozycji przewiduje podzielenie programu aplikacyjnego na tzw. węzły wirtualne (ang. virtual nodes) i ograniczenie komunikacji międzywęzłowej tylko do przesyłania parametrów wejścia zadania.

Tworzenie węzłów wirtualnych może polegać na wyodrębnieniu zadań w grupy. Jedyną dozwoloną postacią komunikacji między takimi węzłami wirtualnymi jest spotkanie zadań. Tworzone dynamicznie zadania są wykonywane w węźle macierzystym (ang. parent). Węzły wirtualne są statycznie przypisane do węzłów fizycznych. Węzłami fizycznymi mogą być pojedyncze procesory lub podzbiory połączonych procesorów systemu wielokomputerowego. Procesory węzłów fizycznych muszą być ściśle powiązane przez pamięć wspólną, aby uzyskać dostęp do współdzielonych danych. Oprogramowanie dla węzła jest kompilowane przez kompilator dla rozproszonych obiektów docelowych według opisu podanego w poprzednim punkcie. W takim ujęciu dane mogą być dzielone wewnątrz węzła wirtualnego, pod warunkiem, że znajdują się w pamięci wspólnej. Procesory tworzące węzeł wirtualny muszą być jednorodne, aby uniknąć konfliktów między różnymi reprezentacjami danych w pamięci wspólnej.

W środowisko programowe Ady (APSE) powinny być włączone automatyczne narzędzia konstrukcyjne. Muszą one dokładnie rozumieć sprzęt docelowy. Pozwoli to na automatyczne generowanie kodu wynikowego zależnie od danego systemu docelowego.

Dodatnią stroną tej metody jest zezwolenie na późne wiązanie węzłów wirtualnych z węzłami fizycznymi. Dzięki temu projektant może abstrahować od fizycznych połączeń w zestawie. Szczegóły konfiguracji fizycznej są potrzebne dopiero w końcowej fazie cyklu projektowego.

ROZPROSZENIE KODU ŹRÓDŁOWEGO

Wymienione podejścia mają tę wadę, że cały ciężar implementacji docelowego systemu rozproszonego spoczywa na twórcy kompilatora. Alternatywę stanowi podzielenie kodu źródłowego i kompilowanie go dla konkretnych obiektów docelowych przy użyciu istniejących kompilatorów. Takie podejście jest obecnie przedmiotem badań, jako część metodologii programowania systemów wielokomputerowych.

Celem tych prac jest ułatwienie programowania w Adzie zestawów wielokomputerowych, przy uwzględnieniu minimalnych ograniczeń projektowych. Dla systemu wielokomputerowego opracowuje się wielozadaniowy program w środowisku programowym Ady (APSE). Projektowanie i implementację oprogramowania wykonuje się przez jego scalanie i testowanie w tymże środowisku. Tak więc opracowywanie i testowanie odbywa się niezależnie od rozproszenia oprogramowania na sprzęcie docelowym.

Przetestowane oprogramowanie wielozadaniowe jest następnie dzielone w zależności od architektury systemu wielokomputerowego. Proces ten może być zautomatyzowany za pomocą proponowanego narzędzia nazwanego ASP (ang. Automated Software Partitioner — automatyczny rozdzielacz oprogramowania). Dostęp do rozproszonych zasobów programowych odbywa się przez procedury emulujące te zasoby. Procedury te inicjują komunikację z procesorami zewnętrznymi przez podsystem komunikacji międzyprocesorowej. Wywołania procedur i wywołania wejść są emulowane przez lokalne procedury mające w miarę możliwości te same nazwy i składnię. Przypisania zewnętrznych obiektów i dostęp do atrybutów wymagają pewnej modyfikacji kodu źródłowego, w celu zastąpienia przypisania obiektu przez wywołanie procedury (obiekt:=) i dostępu do atrybutu przez wywołanie funkcji (obiekt'ATRYBUT). Można to osiągnąć za pomocą narzędzia zwanego ASD (ang. Automated Software Distributor — automatyczny dystrybutor oprogramowania).

Oprogramowanie rozproszone można testować w środowisku programowym Ady (APSE). Program główny każdego procesora umieszcza się wtedy w zadaniu i importuje pakiet symulujący sieć komunikacyjną architektury docelowej. Następnie ASD na podstawie oryginalnego kodu źródłowego i specyfikacji użytkownika generuje pliki źródłowe dla pojedynczych procesorów systemu docelowego. Polega to na pobraniu kodu źródłowego przydzielonego każdemu procesorowi, wykonaniu wszystkich potrzebnych modyfikacji i wygenerowaniu wymaganych pakietów emulatorów zasobów. Powstałe pliki źródłowe każdego procesora są kompilowane w środowisku APSE i kod wynikowy jest ładowany do procesorów docelowych w celu zintegrowania ze sprzętem i przetestowania. Cały zestaw może składać się z procesorów

niejednorodnych, ponieważ Ada zapewnia przenośność kodu źródłowego.

Podział oprogramowania w rozproszonych systemach docelowych można ułatwić stosując metodologię projektowania ukierunkowaną obiektowo. Pakiety zewnętrzne są wtedy dostępne przez pakiety odpowiednich emulatorów zasobów. Użycie takiej metodologii może zwiększyć liczbę ograniczeń, które należy uwzględnić podczas dekompozycji systemu, pozwala jednak na rozmieszczenie przezroczystego kodu źródłowego przez zestaw narzędzi ASP/ASD.

METODA ODRĘBNYCH PROGRAMÓW

Alternatywą w stosunku do rozmieszczenia kodu przez kompilator lub procesor jest przydzielenie zasobów programowych we wstępnej fazie projektowania systemu i opracowywanie odrębnych programów. Użycie emulatorów zasobów o takich samych nazwach jak zasoby zewnętrzne pozwala testować APSE i rozproszony system docelowy przy minimalnych modyfikacjach kodu źródłowego.

Komunikacja międzyprocesorowa jest w tym podejściu na ogół ograniczona do przesyłania parametrów wywołań procedur i wejść. Przykładowo, w programie symulacji zestawu silnie powiązanych komputerów oprogramowanie symulowanych procesorów zawiera się w odrębnych zadaniach. Wejścia zadań zewnętrznych koduje się explicite. a sprzęt do komunikacji międzyprocesorowej jest symulowany przez zadania.

Ujemną stroną metody odrębnych programów jest to, że ponowne przydzielenie oprogramowania jest trudne w późniejszych fazach cyklu istnienia.

STOSOWALNOŚĆ ADY W ŚRODOWISKU ROZPROSZONYM

Przy implementacji zarówno kompilatorów dla obiektów rozproszonych, jak i emulatorów zasobów kodu źródłowego występuje szereg trudności związanych z niedostosowaniem Ady do użycia w tych aplikacjach. Niektóre konstrukcje, np. warunkowe wywołania wejść, typy wskaźnikowe i wejście-wyjście, sprawiają szczególne trudności. Powoduje to pewne ograniczenie w opracowywaniu programów, które mogą być rozproszone.

Warunkowe i synchroniczne wywołania wejść zadań zewnętrznych mogą wpływać ujemnie na efektywność wykonania z powodu narzutu komunikacyjnego. Techniki rozpraszania na poziomie źródłowym wymagają modyfikacji kodu źródłowego, ponieważ jego semantyka może być zmieniona przez procedury emulatora.

Użycie typów wskaźnikowych w środowisku rozproszonym jest również kłopotliwe. W czasie wykonania należy określić, czy wartość wskaźnikowa oznacza obiekt lokalny czy zewnętrzny. Jeżeli obiekt wskaźnikowy i desygnowany znajdują się w różnych procesorach, to może wystąpić nadmierny narzut komunikacyjny. Inny problem może pojawić się wtedy, gdy alokatory w różnych procesorach tworzą nowe obiekty. Nowo utworzone obiekty muszą mieć niepowtarzalne wewnętrzne wartości wskaźnikowe. Wyklucza to rozproszenie kodu źródłowego typów wskaźnikowych. Dodatkowe trudności napotyka się wtedy, gdy obiektami desygnowanymi są typy zadaniowe.

Ada nie zapewnia przyporządkowania jednostek kompilacji określonym procesorom. Powoduje to kłopoty z przyporządkowaniem programów obsługi odpowiednim urządzeniom wejścia-wyjścia. Może okazać się bowiem, że niektóre programy obsługi muszą rezydować na więcej niż jednym procesorze. Dodatkowo powstaje więc problem sformułowania reguł zasięgu.

Wykorzystanie przerwań, obsługa wyjątków i zależność maszynowa stanowią następną grupę problemów do rozważenia. Przerwanie występujące w określonym komputerze musi prawidłowo oddziaływać na cały rozproszony system. Nie jest powiedziane, gdzie powinna odbywać się obsługa wyjątków — w procesorze wykonującym kod zgłoszenia wyjątku czy w zewnętrznym procesorze, który wywołał ten kod. Nie wiadomo też, czy obydwa procesory powinny wiedzieć o wyjątku. Specyfikacja reprezentacji i inne cechy właściwe danej maszynie mogą powodować kłopoty, jeśli jednostka kompilacji nie może być przypisana odpowiedniemu procesorowi.

Istotnym problemem w programowaniu systemów rozproszonych jest tworzenie kodu wtórnego. Użytkownik powinien mieć możliwość swobodnego wyboru segmentów kodu, które mogą być powtarzane w kilku procesorach systemu. Segmenty kodu wtórnego mogą być użyte do zredukowania komunikacji międzyprocesorowej. Mogą one zawierać tylko kod wielobieżny, niezmienny względem czasu. Rozproszenie kodu źródłowego wymaga wygenerowania kopii deklaracji typów i kopii jednostek rodzajowych przy rozdziale oprogramowania.

Oprac. M. KUC

Pierwszy mikrokomputer z mikroprocesorem 80386

Nowy komputer osobisty Deskpro 386 firmy Compaq Computer Corp. z Houston zawiera 32-bitowy mikroprocesor firmy Intel 80386. Konstruktorzy tego mikrokomputera zapewniają, że realizuje on większość oprogramowania IBM PC (jak Lotus 1-2-3, dBase 3 i Framework II) 2—3 razy szybciej niż PC/AT. Uzyskuje się tu wydajność do 4 mln rozkazów/s, a przy programach dla komputerów 16-bitowych — 1,5—2 mln rozkazów/s. Dodanie koprocesora 80287 zwiększa wydajność operacji zmiennoprzecinkowych.

Oprócz szybkości, Deskpro 386 ma wiele innych zalet, m.in. większe podsystemy napędów dyskowych, możliwość wykorzystania rezerwowej pamięci kasetowej oraz opcjonalne pakiety rozbudowanej grafiki i monitora kolorowego.

Opracowując ten mikrokomputer firma przestała być jednym z producentów sprzętu zgodnego z IBM PC, lecz zadeklarowała swoją niezależność. Model 40 tego komputera jest sprzedawany za ok. 6500 dolarów, a model 130 za 8800 dolarów. Dla porównania PC/AT kosztuje ok. 5000 dolarów.

Deskpro jest wyposażony w pamięć operacyjną o pojemności 1 MB, z możliwością rozszerzenia do 14 MB. Należało tu wybrać odpowiednio szybką, a jednocześnie niezbyt drogą pamięć, która umożliwiłaby wykorzystanie szybkości mikroprocesora 80386. Analizowano różne systemy i wybrano pamięć statyczną z dekodowaniem kolumn, która pozwala poszczególnym kostkom działać jak pamięć statyczna w tej części cyklu, w której adresowane są kolumny. Kostki takiej pamięci o pojemności 1 MB stały się standardem.

Główny pakiet pamięciowy zawiera kostki 256 × 1 Kb z czasem dostępu 100 ns. Pojemność ta może być zwiększona dwukrotnie przez wetknięcie dodatkowych kostek na pakiecie. Nakładając dodatkowe pakiety na pakiet główny można zwiększyć pojemność do 10 MB. Oferowane są dwa rodzaje pakietów dodatkowych: o pojemności 1—2 MB przy wykorzystaniu kostek 256 Kb i 4—8 MB przy użyciu kostek 1 MB. Dodatkowe 4 MB mogą być dodane przez wetknięcie pakietu w jedną z czterech łączówek zapasowych.

Opracowano dwie nowe stacje dysków stałych. Model 40 jest wyposażony w napęd typu ST506 Winchester o pojemności 40 MB, a model 130 w stację typu ESDI (ang. Enhanced Small Device Interface — usprawniony sprzęg małych urządzeń) o pojemności 130 MB. Pierwsza z tych stacji ma średni czas dostępu poniżej 30 ms i przesyła informację z prędkością 5 Mb/s. Dla drugiej stacji parametry te mają wartości odpowiednio 25 ms i 10 Mb/s. Liczba sektorów na ścieżkę w pierwszym przypadku wynosi 17, a w drugim — 34. Pakiety sterowników

tworzą wraz ze stacjami integralne jednostki, choć poprzednio były one montowane oddzielnie. Obecnie, przez zastosowanie montażu powierzchniowego, udało się zmieścić sterowniki pod stacjami, co poprawia niezawodność i oszczędza łączówkę.

Jak wiadomo, dane z dysku stałego mogą być przechowywane w opcjonalnej stacji kasetowej o pojemności 40 MB. Stosuje się kasety DC 2000 firmy 3M. Przesyłanie danych odbywa się z szybkością 500 KB/s.

Firma współpracowała z Intelem przy opracowywaniu tego mikrokomputera w celu zapewnienia zgodności programowej z pakietami pisanymi dla komputerów z mikroprocesorami 8088, 8086 i 80286.

Mikrokomputer ten jest wyposażony w układy konwersji danych na 32-bitowej szynie, co pozwala na wykorzystanie dodatkowych układów (jak pakiety sieci lokalnych lub wizyjne), projektowanych dla systemów 8- i 16-bitowych. Pakiety takie mogą być wtykane do dodatkowych łączówek, których jest sześć w modelu 40 i pięć w modelu 130.

Kolorowa grafika wysokiej jakości jest uzyskiwana dzięki układowi EGA (ang. Extended Graphics Interface — usprawniony sprzęg grafiki) realizowanemu na kostce firmy Chips Technology Inc. Stosując taki pakiet i opcjonalny monitor kolorowy uzyskuje się jednoczesne wyświetlenie 16 (spośród 54 możliwych) kolorów z rozdzielczością 640 × 350 elementów.

Systemem operacyjnym jest tu MS-DOS wersja 3.1 firmy Microsoft ze zmianami umożliwiającymi rozszerzenie pojemności pamięci ponad granicę 640 KB. System ten jest także dostosowany do 32-bitowej struktury szyny, przy czym większość oprogramowania dla komputerów 16- i 8-bitowych realizuje się tu bez zmian.

Opracowywany jest system operacyjny MS-DOS 5.0, który będzie dostosowany do mikroprocesora 80386 i pamięci o pojemności 16 MB bez układu rozszerzającego. Innym systemem operacyjnym opracowywanym również przez Microsoft jest pochodzący od Unixa system Xenix System V/386, który miał być ukończony w pierwszej połowie 1987 r. Będzie to pierwszy system operacyjny wykorzystujący w pełni 32-bitową architekturę mikroprocesora 80386 z pamięcią wirtualną, większą przestrzenią adresową i stronicowaniem pamięci. Można natomiast bez modyfikacji korzystać z dostępnego już systemu Xenix System V/286.

Użycie mikroprocesora 80386 daje komputerom osobistym możliwości, jakie dotychczas miały minikomputery. Użytkownicy takich mikrokomputerów mogą tworzyć na nich systemy ekspertowe i wykorzystywać je do obliczeń naukowych.

Fakt, że to nie firma IBM jako pierwsza wprowadziła mikrokomputer z mikroprocesorem 80386 oznacza, że obecnie Intel określa standard dla następnej generacji komputerów osobistych. S. Merrin, analityk komputerów osobistych firmy Gartner Group uważa, że odpowiedni komputer IBM nie pojawi się wcześniej niż w końcu 1987 r.

Firma Compaq natomiast, która przez cztery i pół roku adaptowała standardy ustalone przez dwie generacje IBM PC, może się teraz rozwijać jeszcze szybciej. Nie zaniedbuje ona rozwoju innych wyrobów, a więc komputerów przenośnych i poprzednika ostatniej maszyny — mikrokomputera Deskpro 286. Gdy IBM ogłosiła swój XT Model 286 z mikroprocesorem 80286, Compaq odpowiedziała nowszym modelem Deskpro 286 i obniżką cen. Komputer przenośny tej firmy, zgodny z IBM, zajął drugie miejsce wśród mikroprocesorów sprzedawanych handlowcom. Prezes firmy, Rod Canion stwierdził, że Compaq nie usiłuje narzucać standardów w dziedzinie 32-bitowych komputerów osobistych, gdyż zrobiła to firma Intel. Dodał on, że jeśli IBM opracuje swą własną wersję 80386, nie będzie to już Intel 80386.

Deskpro 386 zyskał sobie dobrą opinię w ocenie przedstawicieli firm opracowujących oprogramowanie jak Lotus czy Ashton-Tate, a Intel i Microsoft brały udział w opracowaniu mikrokomputera.

Opracował
JAN RYŻKO
na podstawie „Electronics" No 31, 1986

Najstarsze przedsiębiorstwo polonijne branży elektronicznej

impol II

02 658 warszawa, ul. filona 16, tel. 43 03 84, 437566, 43 93 41, tlx 817218

Specjalizuje się od 5 lat w dostawach sprzętu mikrokomputerowego i oprogramowania:

✓ imp 85 w+ rewelacyjnie tani mikrokomputer z 4 stanowiskami i pamięcią masową 20 MB

✓ imp 85 m+ mikrokomputer z pamięcią operacyjną 0,5 MB i masową 1,5 - 21 MB, najlepszy w swojej klasie

✓ pc 8/16 przystawka zmieniająca Twój 8-bitowy mikrokomputer w 16-bitowy odpowiednik PC za ułamek jego ceny

✓ master najtańszy wielodostęp do PC/AT (4 stanowiska)

✓ imp 86 standard XT Turbo

✓ imp 8500 terminale ekranowe, odpowiedniki VT-52 i ICL 7182/2 (ODRA, ICL)

✓ Oprogramowanie użytkowe na dowolne mikrokomputery

Gwarancja do 1,5 roku. Dostawa i instalacja u klienta. Pełna dokumentacja. Szkolenie techniczne. Doradztwo.

impol II posiada własne opracowania konstrukcyjne dostosowywane do potrzeb odbiorcy oraz wyspe[illegible]zowane biuro programowania. Serwis naszego s[illegible] prowadzą punkty w Warszawie, Bydgoszczy, Gliwicach, Szczecinie i Krakowie.

EO/544/87

Logo Sinclaira i jego modyfikacje — przyczynek do dyskusji o jakości pozasprzętowej bazy szkolnej edukacji informatycznej

Mija drugi rok, odkąd hasło „informatyka w szkołach i dla szkół" rozbrzmiewa donośnie wzdłuż i wszerz kraju. Tyle samo mniej więcej czasu poświęcono na rzecz propagowania w szkolnej informatyce języka programowania Logo. Zarówno nauczyciele jak i informatycy mają już za sobą niejedno wspólne doświadczenie, wymianę poglądów, konferencję i szkolenie. Dotyczy to szczególnie problemów dydaktyki podstaw informatyki i pożytków, jakie niesie dla powszechnej edukacji informatycznej oparcie się na języku Logo.

Niniejszy artykuł stawia sobie za cel ukazanie niedostatków realizacji języka Logo (i związanych z nią spraw innych, np. jakości dokumentacji, możliwości realnych zastosowań), z którymi przychodzi borykać się nauczycielom. Dla lepszego zrozumienia pobudek, które skłoniły nas do podjęcia tematu w ten sposób, czujemy się w obowiązku oznajmić na wstępie o sprawach związanych z Logo, a nie budzących w nas wątpliwości:

- jeśli należy uczyć programowania w szkołach, to niewątpliwie wybór Logo jest ze wszech miar uzasadniony;
- jeśli zakłada się planowy rozwój wyposażenia sprzętowego szkół, to należy liczyć się z długotrwałą egzystencją realizacji Logo opartej na oryginale Sinclair Logo.

Obie tezy przyjmujemy tu bez dodatkowego uzasadnienia, albowiem uzasadniano je przy wielu okazjach i sąd ten w pełni podzielamy. Dodać wypada, że jesteśmy żywo zainteresowani zagadnieniami dydaktyki Logo w szkołach. W instytucji, którą reprezentujemy, opracowano zarówno materiały metodyczne do nauczania Logo [1], jak i podręcznik tego języka na ZX Spectrum [2]. Trudno by więc posądzać nas o negatywny stosunek wobec roli przypadającej Logo w propedeutyce informatyki. Przeciwnie, jesteśmy zwolennikami tego języka ze względu na przejrzystość jego koncepcji i wynikające stąd walory dla pierwszego kontaktu z problemami programowania lub choćby przez wzgląd na propozycję gimnastyki umysłowej, jaką za pomocą Logo można zaproponować młodym użytkownikom mikrokomputerów w przerwach między jednym testem klawiatury a drugim (czyli większością spośród gier komputerowych).

O ZMIANACH I ICH KIERUNKACH

Informatyka nie jest — jak literatura — dziedziną określającą charakter narodów. Przeciwnie, jak inne osiągnięcia nauki i techniki, dociera do najróżniejszych zakątków świata w formie dość jednolitej. Jedną z cech informatyki w sferze oprogramowania podstawowego (nie mylić z oprogramowaniem problemowym, przeznaczonym dla nieinformatyków do bezpośredniego — z korzyścią — użytkowania) jest zakorzenienie się w nim słów (tu znowu podkreślenie, słów — a nie języka) lub skróceń pochodzących z angielskiego. Przyczyny takiego stanu są różne, jedną z nich da się wytłumaczyć na gruncie samej informatyki: oszczędność środków wyrazu, tj. zwięzły alfabet (26 liter) i statystycznie najkrótsze słowa.

Jak wiemy, w odniesieniu do Logo — języka operującego dość znaczną liczbą (150 w Sinclair Logo) słów zastrzeżonych, a mówiąc ściślej: nazw procedur i funkcji pierwotnych — w wielu krajach powstało naturalne dążenie do zastąpienia ich rdzennymi odpowiednikami. Stało się tak i z Logo Sinclaira, które narodziło się w nowej szacie nazw procedur i funkcji pierwotnych pod tytułem „Polskie Logo" [3].

Swego czasu dyskutowano, czy kierunek taki jest w dydaktyce informatyki uzasadniony [4, 5]. Dziś wiadomo, że i tak, i nie. Tak, bo ułatwia zabawę z komputerem najmłodszym uczestnikom informatycznej rewolucji. Nie, bo rodzimą enklawę słownikową porzuca szybko każdy, kto gotów jest poznać i używać jakichkolwiek innych języków programowania. Podkreślmy, że dyskutowana kwestia słownika [6, 7] nie ma nic wspólnego z potrzebą operowania klawiaturą, zawierającą obok standardowych dla informatyki znaków także zbiór liter polskich. Tę uznać należy za ważną zawsze, a w dydaktyce szkolnej z powodów prozaicznych (pochwała ortografii) za niezbędną.

Stało się tedy, że poprzez wymianę 1—2% kodu, Sinclair Logo przemówiło polskim słownictwem. Zważmy jednak, iż z przyczyn zasadniczych wypływających z reguł składniowych Logo, nigdy żadne Logo nie przemówi **po polsku**. Jednocześnie stało się jednak i co innego. Oto — jak wynika z imperatywnego oświadczenia opublikowanego w miesięczniku „Komputer" [3] — wprowadzone zmiany eliminują w Polskim Logo obudowanie zasadniczego modelu interpretera krótkim chociażby programem w języku BASIC. Wyrażono wówczas powątpiewanie o jakiejkolwiek z tego korzyści. Skutek — w Polskim Logo (w obecnej postaci) utracono możliwość prostej adaptacji systemu do współpracy z dyskietką (co w Sinclair Logo jest możliwe). O różnicy jakościowej w pracy z wykorzystaniem dyskietki w stosunku do działań z magnetofonem przekonywać nie będziemy.

Druga sprawa wynikła przy okazji modyfikacji kodu firm SOLI/LSCI (Sinclair Logo), to usunięcie błędów (patrz: punkt następny). Niestety, niewiele spośród istniejących błędów usunięto, niekiedy zresztą nieskutecznie — powstały błędy nowe (wyzwanie błędom realizacji Sinclair Logo rzucono w końcu minionego roku także w Instytucie Informatyki UWr). Dodano natomiast podprogram zamalowywania wnętrz figur. Tu uwaga, że wywołanie dowolnych podprogramów w kodzie wewnętrznym zakłada zarówno wersja z polskim, jak i angielskim słownictwem. Wprowadzono też w Polskim Logo transkrypcję językową sygnałów błędów, co niewątpliwie sprzyja szkolnym celom.

Usunięcie błędów z oprogramowania zwykle bywa trudne. Jest szczególnie trudne w wypadku braku dokumentacji i pracy innego, niż twórcy oprogramowania, zespołu. Sytuacja taka ma miejsce w poczynaniach z Logo Sinclaira. Jest rada, aby się z tym nie borykać. Napisać rzecz od nowa. Zostawmy jednak uwagi (żarty?) na potem.

---

[1]) E. Kołczyk, M. Malicka: Materiały pomocnicze do nauczania języka Logo w szkole średniej. Raport P-29. Instytut Informatyki Uniwersytetu Wrocławskiego, październik 1986.

[2]) E. Gurbiel, H. Krupicka, Z. Płoski (red.): Programowanie i Logo. Wydawnictwo Uniwersytetu Wrocławskiego (w druku).

[3]) Polskie Logo. Komputer, nr 3, 1986, str. 10—12.

[4]) Logo po polsku. Bajtek, nr 4, 1986, str. 5—8.

[5]) M. M. Sysło: List do Redakcji. Biuletyn PTI, nr 5—6, 1986.

[6]) W. Iszkowski: O języku polskim w informatyce ciąg dalszy. Biuletyn PTI, nr 11, 1986.

[7]) M. M. Sysło: List do Redakcji (przekazany do opublikowania w Biuletynie PTI, luty 1987).

O KŁOPOTLIWYM DZIEDZICTWIE

Logo jest piękne! — zakrzykniemy z przekonaniem i argumentami w zanadrzu. Gorzej z jego realizacją. Logo Sinclaira sprawia wrażenie oprogramowania wykonanego pośpiesznie (czyżby w myśl dewizy: szybko zrobić, szybko zapomnieć?). Niektóre z błędów tkwiących w tej realizacji powodują przysłowiowe jeżenie się włosów na głowie. Logo Polskie — w wersji rozprowadzanej obecnie — niestety, niewiele odbiega pod tym względem od pierwowzoru.

Zanim wymienimy przykładowe błędne reakcje Logo Sinclaira dziedziczone w Polskim Logo podnieśmy problem z całą ostrością: oprogramowanie dydaktyczne, bądź do dydaktyki służące, z którym styka się początkujący, winno być **absolutnie niezawodne.** Idzie tu o dwie sprawy:
— nieprowadzenie nauki (wszystko jedno — programowania czy czego innego) na manowce technicznych usterek narzędzi dydaktycznych,
— niewyrabianie (przecież niezgodnego z ogólnoinformatyczną rzeczywistością i posłannictwem) mniemania, że komputer jest równie niekonsekwentny, jak jego twórca.

Oto wybór błędów w Sinclair vel Polskim Logo wedle oprogramowania zawartego na kasecie kupionej w pobliskim „empiku".

Przykład 1. Pozostały nadal w tej realizacji (POLLOG 2.2) usterki związane z konwersją słów, które są liczbami. Zilustrujemy to na przykładzie kilku wyrażeń i instrukcji.

1. Możemy zobaczyć, że wartością wyrażenia:

SŁOWO "A "032

jest słowo A032, natomiast wyrażenie:

SŁOWO "A BEZPIERW "4032

ma wartość równą słowu **A32** (a nie **A032**, jak można by oczekiwać).

2. Instrukcja:

PP [7 NP 1E2]

wydrukuje trzy słowa: 7 NP 100 — traktując dwa pierwsze elementy listy literalnie i przekształcając postać liczby 1E2 (co jest sprzeczne z zasadą literalnego traktowania wszystkich elementów na liście). Z kolei, instrukcja:

PP PIERW [1.2.3]

wydrukuje 1.2 i pominie (!) .3 ze słowa będącego elementem listy.

3. Zdefiniujmy procedurę:

```
OTO CIEKAWA_PROCEDURA :a
JEŚLI PUSTE? :a [STOP]
PP :a
CIEKAWA_PROCEDURA BEZPIERW :a
JUŻ
```

Wykonajmy następnie instrukcje:

```
CIEKAWA_PROCEDURA "e1e2e
CIEKAWA_PROCEDURA "e1e2e3
```

Efekt ich działania będzie następujący — w pierwszym wypadku:

```
e1e2e
1e2e
e2e
2e
e
```

a w drugim wypadku, sprzeczny z oczekiwanym:

```
e1e2e3
100
0
```

4. Poprzednie błędy dotyczyły tylko kłopotów ze słowami „liczbopodobnymi" w literalnych kontekstach list Logo. Błąd polegający na zmienianiu tekstów leżących w listach na prawo od innych list, jak w przykładzie:

?PP [Powiedz Jasiu, powiedz Olu, [gdzie tak dobrze] jak w przedszkolu?]

Powiedz Jasiu, powiedz Olu, gdzie tak dobrze JAK W PRZEDSZKOLU?

uniemożliwia w praktyce każdą mniej banalną próbę operowania tekstami z użyciem list, tych podstawowych w Logo struktur danych.

Przykład 2. Interpreter polskiej wersji Sinclair Logo „nie lubi" jako elementów listy słów pierwotnych poprzedzonych cudzysłowem (operatorem dosłowności). Po zdefiniowaniu procedury:

```
OTO PRZYKŁAD
PP :a
PP PIERW :a
JUŻ
```

i po przypisaniu zmiennej a wartości:

PRZYP "a ["OTO jest]

wykonanie procedury PRZYKŁAD spowoduje wydrukowanie elementów listy:

"OTO jest

oraz — zupełnie nieoczekiwanie — wydrukowanie w ciągu:

"S??$ SE

lub innego, albo zawieszenie działania interpretera, uniemożliwiające kontynuowanie pracy bez ponownego załadowania [8]).

Przykład 3. Ten przykład, dla odmiany, obrazuje pozorne poprawienie błędu w polskiej edycji interpretera Logo. Usunięto błąd, zastępując go innym. Oto protokół z konwersacji:

```
?POKAŻ LISTA "abc LISTA "DEF [ ]
[abc [def [ ] ] ]
```

ale już instrukcja:

POKAŻ (LISTA "abc "def [ ])

spowoduje wydrukowanie listy dwuelementowej:

[[abc def] [ ]]

zamiast oczekiwanej listy:
[abc def [ ]]

złożonej z trzech elementów. W wersji oryginalnej Sinclair Logo, instrukcja:

SHOW LIST "abc LIST "def [ ]

powodowała sygnalizację błędu. Natomiast instrukcja:

SHOW (LIST "abc "def [ ])

działała poprawnie.

Przykład 4. Niedomagania nie omijają nawet instrukcji graficznych. Żółw w dotychczasowych mutantach Sinclair Logo wykazuje „odchylenie prawicowe". Po wykonaniu ciągu instrukcji (kąt większy od 360°):

CS LW 389

żółw spogląda w tę samą stronę, co po wykonaniu:

CS PW 389

Przykład 5. Ostatni przykład przedstawia błędy interpretera, które są nie tylko nieprzyjemne w skutkach, ale wręcz antydydaktyczne. Okazuje się, że jeśli a i b są liczbami, to nie zawsze $a < b$ implikuje $b > a$. Uczeń, programując w Sinclair Logo (również w wersji polskojęzycznej), „dowiaduje się", że nierówności: $256 > 0$ i $1 > .5$ itp., są fałszywe. Ilustruje to poniższy zapis fragmentu pracy w Polskim Logo:

```
?PP .4 < 1
PRAWDA
?PP 1 > .4
PRAWDA
```

[8]) Z. Kasprzycki: Zachłanne instrukcje w polskim Logo. Komputer, nr 1, 1987, s. 28—29.

```
?PP .5 < 1
PRAWDA
?PP 1 > .5
FAŁSZ
?PP 128 > 0
PRAWDA
?PP 256 > 0
FAŁSZ
?PP 512 > 0
FAŁSZ
```

Co teraz ma zrobić nauczyciel informatyki (lub matematyki)?

O ZNACZENIU SZCZEGÓŁU W DOKUMENTACJI I PODRĘCZNIKU

Język Logo zwykło się traktować jako szczególnie łatwy w nauczaniu. Zapewne taki jest, zwłaszcza gdy rozważymy pewien jego podzbiór, a już z pewnością umożliwia łatwe tworzenie niewielkich konstrukcji, czemu sprzyja jego ogólne środowisko programowe — łatwość przechodzenia od opisywania procedur do ich uruchamiania i testowania. Dotyczy to także omawianych wersji. Czy jednak łatwość pierwszego „twórczego" kontaktu z Logo ma oznaczać, że wykład bądź podręcznik Logo (zwłaszcza ten ostatni, przeznaczony dla uczniów i nauczycieli szkoły średniej, a nie dla uczniów szkoły podstawowej) ma się ograniczać tylko do omówienia elementów języka Logo metodą poglądową? Zapewne nie. Nie, ponieważ taki wykład, kurs, a przede wszystkim podręcznik nie da możliwości rzeczywistego odnalezienia się w szczegółach każdego języka programowania, co dotyczy w równym stopniu Logo. Nie można uczyć języka programowania za pomocą komputera wyłącznie metodą prób i błędów. Znajomość **ogólnych koncepcji** języka nie wystarcza, niestety, by skutecznie się nim posługiwać. Odnosi się to — rzecz znamienna — na równi do języków sztucznych i naturalnych. Z kolei, kiedy w grę wchodzi opis szczegółowy, powstaje wtórnie problem językowy — jak mu podołać? W wypadku języków programowania rozwiązanie jest znane od dawna — gramatyka. Czy jest uzasadnione opracowanie — choćby przybliżenia — gramatyki w odniesieniu do Logo? Wymienimy znowu dwa powody:

1. W upowszechnianych realizacjach Logo, wzorowanych na Logo Sinclaira, trudno mówić o zastosowaniach (na przykład do budowy oprogramowania dydaktycznego lub narzędziowego), dokładniej — są one mocno ograniczane nie przez koncepcje języka, lecz przez powolność interpreterów i wielce skromne rozmiary dostępnej dla programów pamięci. Skoro zatem, mówiąc o konkretnych realizacjach Logo, pozostaje wyraźnie w polu widzenia aspekt nauczania koncepcji programowania jako takiego, stanie się dobrze, gdy i metody nauczania będą od początku zgodne z tematyką. Możliwość częściowej choćby formalizacji pojęć języka Logo za pomocą gramatyki dobrze służy temu przedsięwzięciu w wypadku nauczania Logo uczniów szkoły średniej w ramach elementów informatyki.

2. Rezygnacja z metody (w części) formalnej w podręczniku omawiającym właściwości języka programowania, przy wielu zaletach, takich jak przystępność, większa atrakcyjność, pobudzanie zainteresowania — zawodzi w sytuacjach praktycznego wykorzystania (weryfikowania) zawartych w podręczniku informacji.

Oto kilka przykładów dla zilustrowania tej tezy, zaczerpniętych z jednego ze skryptów o Sinclair Logo [9]):

1. Z werbalnej definicji liczby wynika, iż w jej zapisie nie może pojawić się znak + (plus). Tymczasem, praktyka programowania wykazuje, że nie tylko można użyć znaku + w liczbie, w wypadku liczby postaci mEc, na przykład **1E+9**, ale interpreter sam dodaje znak + po **E**, jeśli c jest liczbą dodatnią. Na przykład liczba zapisana jako **1.25E12** będzie przedstawiona w postaci **1.25E+12**.

2. W tymże skrypcie użyto określeń „poprawnie zbudowane wyrażenie" lub „prawidłowo zbudowane wyrażenie" bez podania definicji „poprawnie zbudowanego wyrażenia". Zapisano tamże: „Jeśli dwa dowolne, prawidłowo zbudowane wyrażenia, przyjmujące wartości liczbowe napiszemy obok siebie, przedzielając jednym ze znaków (<, >, =, przyp. aut.), to otrzymamy wyrażenie przyjmujące wartość TRUE (prawda) lub FALSE (fałsz) zależnie od wyniku porównania". Wystarczy napisać obok siebie dwa wyrażenia: **RANDOM 3** oraz **0** i przedzielić je znakiem relacji, aby przekonać się, że wystąpi błąd, a nie **TRUE** lub **FALSE**, jak napisano, komentując przykład:

RANDOM 3 > 0.

3. Przytoczymy definicje słowa i listy:

„Słowo w sensie Logo jest to ciąg dowolnych znaków, który nie zawiera:
— nawiasów kwadratowych,
— odstępów,
— znaków + — * / = < > ( ) { } nie poprzedzonych bezpośrednio przez " lub | . " [10])

„Lista jest to ciąg słów lub list ujęty w nawiasy kwadratowe alba para nawiasów kwadratowych".

Z powyższej definicji słowa wynika, że pojedynczy nawias okrągły oraz znak + (podobnie jak inne operatory działań arytmetycznych i relacji) nie są słowami. Skoro jednak **[(a + b)]** jest listą o pięciu elementach (instrukcja **PR COUNT [(a + b)]** wydrukuje liczbę 5), to czym są znaki **(**, **+** i **)**? Niewątpliwie nie są listami. Dodatkowo wartością wyrażeń, WORDP FIRST **[+]**, **WORDP FIRST [(]** i **WORDP FIRST [)]** jest **TRUE.**

Nie jest jasne, dlaczego — zgodnie z cytowaną definicją — nawiasy { i } nie mogą wystąpić w słowie? Wszak instrukcja:

PR COUNT "CZY {TO} JEST {SLOWO}

wydrukuje liczbę **18**, traktując oba użyte nawiasy na równi z literami.

OCENA SYTUACJI „NA ODCINKU LOGO"

Poddajemy temat jakości szkolnego oprogramowania podstawowego pod uwagę i dyskusję Czytelników nie po to, aby Ich nastawić, a zwłaszcza nauczycieli, niechętnie do Logo w jego obecnej postaci. Czynimy to dla przestrogi przed nadmiernym optymizmem, któremu można ulec kojarząc treści programu nauczania pt. „Elementy informatyki" (w jego aktualnym brzmieniu [11]) z emanującą z niego, słabo zawoalowaną namową do użycia (i słusznie!) Logo jako bazy dla materializowania tych treści. Dotyczy to zwłaszcza prób stosowania omawianych interpreterów Logo do celów innych niż nauczanie programowania (w samym Logo tkwi jeszcze inne „niebezpieczeństwo" — zagubienia się w formie (grafika żółwia), a przecież propedeutyka informatyki ma przygotowywać nie tylko do obcowania ze zjawiskiem spełniania bądź wyrabiania przez komputery nowych kanonów estetycznych). W tej ostatniej mierze jest potrzebna — z konieczności — wielka ostrożność wypływająca ze świadomości małej niezawodności oprogramowania służącego za materiał do nauki i ćwiczeń.

Za pomocą Logo Sinclaira i jego pochodnych — nawet przy wszystkich jego mankamentach — można z powodzeniem przekazać podstawowe koncepcje programowania i ... ostrzec uczniów przed skutkami niesolidnego podchodzenia do informatycznej roboty.

Informatyka to, między innymi, umiejętność dokładnego określania. Odnosi się to także do podręczników. Sądzimy, że — również w przypadku Logo — aspekt ten nie powinien zostać zbagatelizowany. Dokładna „specyfikacja" jest niezbędna nie tylko przy budowaniu programu, ale — w nie mniejszym stopniu — przy nauce programowania. W podręczniku „Programowanie i Logo", który być może

[9]) S. Waligórski: Logo na Sinclair Spectrum dla zaawansowanych. Instytut Kształcenia Nauczycieli, Warszawa 1986.

[10]) Ze względów technicznych zamiast ukośnika (ang. backslash) używamy pionowej kreski (red.).

[11]) „Elementy informatyki" — program liceum ogólnokształcącego — Ministerstwo Oświaty i Wychowania, Warszawa, 1985 (zatw. pod nr OP 23-4120-41/85).

wyjdzie spod prasy wraz z ukazaniem się tego tekstu, staraliśmy się (oby skutecznie!) urzeczywistnić nasz pogląd na prezentowanie wiadomości o języku Sinclair Logo.

Pozostaje jednak otwarta kwestia niezawodnego interpretera Logo dla polskich szkół. Obecnie mamy do zaproponowania w IIUWr wersję Sinclair Logo 1.6p (opracowaną przez inż. Adama Początka) ze zmniejszoną liczbą błędów, możliwością używania polskich liter (ale również i dyskietek), spolszczonymi sygnałami błędów i ... zachowanym oryginalnym nazewnictwem procedur i funkcji Logo. **Ponadto, wersja ta jest szybsza od oryginalnej średnio o 25—30% (oszczędności czasu w przykładzie z przybliżeniem okręgu wielokątem o 360 bokach sięga nawet 50%).** Inne możliwości rysują się następująco:
— albo napisać interpreter od nowa (—),
— albo połączyć działania w celu jak najlepszego „odpluskwienia" dostępnych realizacji (—),
— albo wprowadzić do programu pt. „Elementy informatyki w szkołach średnich", po małym (w granicach bezpieczeństwa wytyczonych przez obarczone błędami realizacje) rozpoznaniu koncepcji (10—15 godzin) na gruncie Logo, nauczanie oparte na innym języku (Pascal?),
— albo ... zająć się w szkołach propedeutyką informatyki w wersji konsumpcyjnej, tj. korzystać z usług gotowych programów, miast uczyć programowania; z tym, że wtedy wypadałoby jeszcze poczekać (!) — jak długo?!

EWA GURBIEL, HELENA KRUPICKA, ZDZISŁAW PŁOSKI
Instytut Informatyki
Uniwersytetu Wrocławskiego

Szanowna Redakcjo!

Nie należę do stałych czytelników Waszego pisma. Ostatnio jednak pokazano mi numer 3'1987, w którym na str. 2 rozpatruje się kwestię, czy *logic programming* należy oddawać jako *programowanie w logice* czy *programowanie logiczne*. Optując stanowczo za tym pierwszym rozwiązaniem chciałbym zwrócić uwagę na nietrafność przedstawionych przez Was argumentów.

Programowanie w logice jest nie bardziej żargonowe niż *programowanie w Fortranie* — nie mówimy wszak *programowanie fortranowe*. Natomiast po angielsku używa się zwykle *Fortran programming*, i stąd *logic programming* dla podkreślenia, że *logic* spełnia tu rolę języka programowania.

Z kolei *programowanie logiczne* jest ścisłym odpowiednikiem terminu *logical programming*. Zarówno po polsku jak i po angielsku programowanie inne niż logiczne to programowanie nielogiczne. Używając takiego terminu dawalibyśmy więc do zrozumienia, że ten, kto nie programuje w Prologu, bezsensownie programuje!

Ze względu na duży wpływ Waszego pisma na stan świadomości informatyków w naszym kraju byłbym wdzięczny za opublikowanie tego listu.

Z poważaniem

FELIKS KLUŹNIAK
Instytut Informatyki
Uniwersytet Warszawski

Szanowny Panie,

Dla redaktora nie ma bardziej bałamutnej wiadomości niż to, że jego pismo ma *duży wpływ na stan świadomości informatyków w naszym kraju* — zwłaszcza zawodowej. Czyż jest to możliwe przy ośmiotysięcznym nakładzie oraz bełkocie wylewającym się na ten kraj z łamów miesięczników popularnych o łącznym nakładzie 100 razy większym od INFORMATYKI? Pozostaje tylko wierzyć, że choć część środowiska zawodowego nie utraci instynktu samozachowawczego.

Przechodząc do rzeczy, pragnę odpowiedzieć na Pana argumenty.

Terminu *logika* nie można traktować w pełni analogicznie do nazwy języka programowania, ponieważ trudno zgodzić się ze stwierdzeniem, że logika jest językiem. Logika jest nauką. Jeśli uznano, że zawarty w niej formalizm odgrywa rolę języka programowania, to należałoby mówić *programowanie w języku logiki*, podobnie jak na przykład — *formułowanie zdań w języku matematyki* itp. Skróconie tak powstałego wyrażenia *programowanie w języku logiki* do wyrażenia *programowanie w logice* pachnie błędem językowym. Nie mogę więc uznać pierwszego z Pana argumentów, bo jest oparty — Pan wybaczy — na fałszywej przesłance.

Jeżeli chodzi o drugi argument, to jest on tylko jednym ostrzem broni obosiecznej. Rozumując na gruncie języka potocznego, tak jak w Pana uzasadnieniu nietrafności terminu *programowanie logiczne*, sformułowanie *programowanie w logice* należałoby odczytać dosłownie, tzn. *programowanie w dziedzinie nauki zwanej logiką*, co oczywiście jest niezgodne z intencją autorów tego sformułowania. Na płaszczyźnie języka potocznego nie można więc prowadzić skutecznej analizy znaczeń terminów i należy to podejście odrzucić.

Ważniejsze wydaje mi się co innego. Zwrot *programowanie logiczne* jest terminem jako całość i nie można oddzielać od niego przymiotnika *logiczny*. Dlatego nie istnieje desygnat nazwy *programowanie nielogiczne*, tak jak nie ma desygnatu nazwy *programowanie niematematyczne* (choć istnieje termin *programowanie matematyczne*). Co więcej, czy komuś znającemu termin *układ logiczny* przychodzi do głowy szukanie *układu nielogicznego*? Nie ma więc obaw, że ktoś opacznie zrozumie *programowanie logiczne* w tekście technicznym. Nie mogę zatem także przyjąć Pana drugiego argumentu.

Cóż z tego, skoro część fachowców może nadal posługiwać się w żargonie wyrażeniem *programowanie w logice*, uważając że logika jest językiem. Wyraz logika ma wszak inne znaczenia, znaczy również tyle co rozsądek, jest też używany w technice w znaczeniu *zespół układów elektronicznych*. Jeśli tak rzeczywiście będzie, to powstaną nowe wątpliwości. Jedna spośród nich wiąże się z pytaniem, czy to wyrażenie jest terminem? Zwrot *programowanie w Fortranie* (ang. Fortran programming) terminem na pewno nie jest, natomiast *logic programming* ponad wszelką wątpliwość jest. Zatem *programowanie w logice* byłoby terminem trzywyrazowym i w dodatku z przyimkiem w środku. Nie trzeba uzasadniać, że z terminologicznego punktu widzenia jest to konstrukcja dość karkołomna i trudno byłoby zalecić jej używanie.

Na zakończenie chciałbym zaznaczyć, że choć odmawiam racji Pana argumentom, co starałem się uzasadnić, to nie uważam, aby moje rozstrzygnięcie było *jedynie słuszne*. Na szczęście język naturalny jest faktem społecznym (a nie matematycznym lub logicznym) i jeśli dostarczy Pan własny artykuł zawierający użycie (niepoprawnego) zwrotu *programowanie w logice*, to zaręczam, że zostanie to uznane.

Z poważaniem

JANUSZ ZALEWSKI
z-ca redaktora naczelnego

Grudziński W.: Zastosowanie Prologu w bazach danych (1)

INFORMATYKA 1987, nr 8, s. 1

Pierwsza część artykułu, zawierająca omówienie istoty programowania logicznego oraz podstaw języka Prolog.

Grudziński W.: Prolog application in data bases (1)

INFORMATYKA 1987, No. 8, p. 1

First part of the paper, which contains a discussion on logical programming essentials and Prolog language principles.

Grudziński W.: Prolog-Anwendung in Datenbanken (1)

INFORMATYKA 1987, Nr. 8, S. 1

Erster Teil des Artikels, der eine Besprechung von Wesen des logischen Programmierung sowie Grundlagen der Prolog-Sprache umfasst.

Bielecki J.: Podstawy grafiki w języku Turbo Pascal (1)

INFORMATYKA 1987, nr 8, s. 5

Pierwsza część omówienia trybów współpracy z ekranem w języku Turbo Pascal na komputerze IBM PC z podaniem przykładowych programów dla podstawowych funkcji.

Bielecki J.: Graphics essentials in Turbo Pascal (1)

INFORMATYKA 1987, No. 8, p. 5

First part of discussion on IBM PC cooperation mode with display in Turbo Pascal, illustrated by exemplary programs for basic functions.

Bielecki J.: Grafikgrundlagen in Turbo Pascal (1)

INFORMATYKA 1987, Nr. 8, S. 5

Erster Teil einer Besprechung von Zusammenarbeitmodi zwischen IBM PC und Bildschirm in Turbo Pascal mit Angabe von Beispielprogrammen für Grundfunktionen.

Shaw M. M.: Poza programowanie wielkoskalowe — kolejne wyzwanie dla inżynierii oprogramowania (2)

INFORMATYKA 1987, nr 8, s. 8

Druga część artykułu, zawierająca charakterystykę zespołów zagadnień inżynierii oprogramowania, wymagających rozwiązania w najbliższych latach.

Shaw M. M.: Beyond programming-in-the-large — next challenges for software engineering (2)

INFORMATYKA 1987, No. 8, p. 8

Second part of the paper, which contains characteristics of software engineering problem set which must be solved in next years.

Shaw M. M.: Ausserhalb der Grossmassstabprogrammierung — nächste Herausforderung für die Software-Engineering (2)

INFORMATYKA 1987, Nr. 8, S. 8

Zweiter Teil des Artikels, der eine Charakteristik von Software-Engineering-Problemen für die nächste Jahre umfasst.

Tadeusiewicz R.: Głosowe wprowadzanie informacji do komputera (2)

INFORMATYKA 1987, nr 8, s. 12

Druga część artykułu, zawierająca omówienie zestawu operacji i przekształceń sygnału mowy podczas jego wprowadzania, przetwarzania i rozpoznawania w komputerze.

Tadeusiewicz R.: Voice information input for computer (2)

INFORMATYKA 1987, No. 8, p. 12

Second part of the paper, which contains discussion of voice signal operations and conversions during input, processing and recognition in computer.

Tadeusiewicz R.: Eingabe phonetischer Information in einem Computer (2)

INFORMATYKA 1987, Nr. 8, S. 12

Zweiter Teil des Artikels, der eine Besprechung von Operationen und Umwandlungen des Sprachsignals während seiner Eingabe, Verarbeitung und Erkennung in einem Computer umfasst.

Raznowiecki A., Syfert A., Zalewski J.: Struktura systemu operacyjnego PC-DOS (2)

INFORMATYKA 1987, nr 8, s. 16

Druga część charakterystyki systemu operacyjnego PC-DOS, zawierająca omówienie sposobu opisu plików.

Raznowiecki A., Syfert A., Zalewski J.: Structure of the PC-DOS operating system (2)

INFORMATYKA 1987, No. 2, p. 16

Second part of the PC-DOS operating system characteristics, which contains discussion of the file description method.

Raznowiecki A., Syfert A., Zalewski J.: Struktur des PC-DOS-Betriebssystems (2)

INFORMATYKA 1987, Nr. 8, S. 16

Zweiter Teil einer Charakteristik von PC-DOS-Betriebssystem, der eine Besprechung von Dateibeschreibungsmethode umfasst.

Bielecki J.: Język C — definiowanie funkcji ze zmienną liczbą argumentów

INFORMATYKA 1987, nr 8, s. 19

Charakterystyka jednego z rozszerzeń języka C, zaimplementowanego w wersji 4.0 kompilatora firmy Microsoft.

Bielecki J.: C language — a definition of functions with variable number of operands

INFORMATYKA 1987, No. 8, p. 19

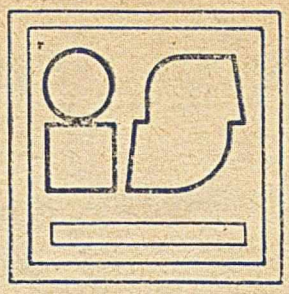
Characteristics of the one from among C language extensions, implemented in Microsoft compiler version 4.0.

Bielecki J.: C-Sprache — eine Definierung der Funktionen mit variablen Operandenzahl

INFORMATYKA 1987, Nr. 8, S. 19

Eine Charakteristik einer von C-Sprache Erweiterungen, die in 4.0-Version des Microsoft-Compilers implementiert wurde.

INTERSOFT



Sp. z o.o.

Al. Ujazdowskie 18 m. 14, 00-478 Warszawa
Punkt handlowy ul. Zamenhofa 4 m. 32, 00-160 Warszawa, tel.: 316-322 lub 280-176

o f e r u j e:

WYBRANE POZYCJE DOKUMENTACJI W JĘZYKU POLSKIM DO KOMPUTERÓW IBM

— Przewodnik programisty IBM
— Sidekick
— Wstęp do grafiki komputerowej (Basic, Turbo Pascal, TG, aut. J. Bielecki) w komplecie 4 dyskietki z przykładami i oprogramowaniem pomocniczym
— Autocad (opr. Computex)
— Symphony (opr. Computex, tylko łącznie z programem)
— Lotus 1-2-3
— Framework
— System operacyjny DOS 2,0 (2 tomy, opr. Computex)
— System operacyjny DOS 3,1 (1 tom)
x — System operacyjny DOS 3,2 (3 tomy)
x — GW-Basic, interpreter (opr. Computex)
— GW-Basic, kompilator (opr. Computex)
x — GW-Basic, kompilacja i konsolidacja programów (opr. Computex)
x — Język "C", opis (opr. Computex)
— Język "C", kompilator (Lattice, opr. Computex)
— Język "C" (aut. J. Bielecki)
— Fortran 77, opis języka (opr. Computex)
— Fortran 77 (aut. J. Bielecki)
— Fortran 77, kompilator (opr. Computex)
— Fortran 77, podstawowa biblioteka numeryczna (opr. Computex)
— Agraph, biblioteka graficzna dla języków Fortran i Pascal (opr. Computex)
— Turbo Graphics (opr. Computex)
— Turbo Pascal, podręcznik użytkownika (opr. Computex)
— Turbo Pascal (aut. J. Bielecki)
— dBase III, programowanie i opis komend (2 tomy)
— Clipper, kompilator do dBase III
x — dBase III, opis komend z przykładami
— Przewodnik zaawansowanego programisty do dBase II/III
— dBase III+, programowanie
— dBase III+, samouczek
x — dBase III+, zastosowania
x — Instrukcja obsługi PC1512 (2 tomy)
x — Instrukcja obsługi dysku twardego do PC1512
x — PC1512, opis techniczny
x — Instrukcja do drukarki Star Gemini 160
x — Chi writer
x — PC writer
— Turbo "C"

Uwaga: pozycje oznaczone "x" są w przygotowaniu. Firma INTERSOFT jest oficjalnym dostawcą oprogramowania i dokumentacji firmy Computex

WYBRANE POZYCJE DOKUMENTACJI W JĘZYKU POLSKIM DO KOMPUTERÓW AMSTRAD

— Instrukcja do komputera CPC 464
— Instrukcja do komputera CPC 6128
— Instrukcja do komputerów PCW 8256/8512, wstęp
— Instrukcja do komputerów PCW 8256/8512, Locoscript
— Instrukcja do komputerów PCW 8256/8512, CP/M
— Instrukcja do komputerów PCW 8256/8512, Mallard Basic
— CPC Basic
— Systemy operacyjne CPC
— Intern 664/6128
— DDI-1 Firmware
— CPC 464 Firmware
x — Ins&Outs
— AutoCad (CPC, PCW)
— Instrukcja do drukarki DMP-2000
— Wordstar
— Protext
— dBase II
x — Praktyczne zastosowania dBase II
— Dr Draw
— Pascal MT+
— Turbo Pascal
— Profi Painter
x — C Basic (3 tomy)
— Fortran 80

Uwaga: pozycje oznaczone "x" są w przygotowaniu.

EO/782/87

KONIEC TWOICH PROBLEMÓW

Nareszcie wszystkie potrzebne Ci dane dotrą na czas w przejrzystej formie tabel i wykresów.

Pakiet oprogramowania na mikrokomputery 16-bitowe

MEGA — BANK

to nowe MEGA możliwości jakie otwierają się przed Twoim przedsiębiorstwem.

Wyobraź sobie

MILIARD

rekordów, które możesz zapełnić według własnych potrzeb i uznania.

Miliard kooperantów, miliard pracowników, miliard produktów — z tym wszystkim nasz MEGA-BANK poradzi sobie bez trudu.

System jest łatwy w obsłudze i opracowany w języku polskim.

Gwarantujemy satysfakcję!

COMPUTER STUDIO KAJKOWSCY

PROFESJONALNE OPROGRAMOWANIE MIKROKOMPUTERÓW

ul. Balladyny 3B, 81-524 Gdynia, tel.: 29-00-18, 24-01-50

EO/74/87